Writing Efficient FORTRAN

BY [redacted] (b)(3)-P.L. 86-36

Unclassified

This paper describes a number of methods that a programmer can use to write more efficient programs. Its goal is to foster the attitude that the efficiency of a program is often as important as its accuracy, and to provide the understanding necessary to accomplish this.

I. INTRODUCTION

This paper has been written in an attempt to influence FORTRAN programmers to write more efficient programs, with little extra effort on their parts. Programming is very much a matter of style, and one's style may be good or bad. Just as one's prose style can be improved through guidance and practice, so too can one's programming style. This paper contains a list of "do's and don'ts" of programming which, if adhered to, will guarantee the development of a more efficient programming style.

The emphasis of the paper is on FORTRAN, and many of the comments apply only to FORTRAN, and, in particular, the CDC 6600 IDASYS version of FORTRAN. Other comments apply more generally, to all higher level algorithmic languages; some, finally, apply to any programming language. The examples employed will be exclusively in FORTRAN.

There are two cardinal rules of efficient programming:

(1) Never perform an unnecessary operation.

(2) If an operation may be performed several ways, always choose the cheapest way.

The remainder of the paper is essentially an explication and expansion of these two principles.

II. SUBSCRIPTED VARIABLES

A programmer should be aware of the effect of his use of a subscripted variable. Variables in general are simply names of storage locations in which values are stored. During compilation, each variable name is replaced with the address of the corresponding storage location. The address is then available during the execution of the program and need not be recalculated.

This is not true for a subscripted variable. Only the address of the first location of the array can be prepared during calculation. But every execution-time reference entails a computation of the effective address.

For a singly dimensioned array, this operates as follows:

address (A(I)) = address (first location of A) + I − 1

For multiply-dimensioned arrays, the situation is somewhat more complex. Let A be dimensioned:

```
DIMENSION A(L,M)
```

Then we have:

address (A(I,J)) = address (first location of A) + I − 1 + L * (J − 1)

Finally, the three-dimensional formulae are:

```
DIMENSION A(L,M,N)
```

address (A(I,J,K)) = address (first location of A) + I − 1 + L * [J − 1 + M * (K − 1)]

In other words, for each additional dimension above one, there is an additional multiplication and addition *for each reference* during execution.

To understand the effects of this extra work, consider the following loop example:

```
      DIMENSION A(10,10)
      DO 10 I=1,10
      DO 10 J=1,10
   10 SUM=SUM + A(I,J)
```

This short loop involves 100 references to the array A, at a cost of one multiplication and one addition each. Now consider the following alternative:

```
      DIMENSION A(10,10)
      DO 10 J=1,10
      JJ=10*(J-1)
      DO 10 I=1,10
   10 SUM=SUM + A(I+JJ)
```

We now have eliminated the 100 multiplications from the array references and replaced them with 10 set-up multiplications. We can improve things even further:

```
      DIMENSION A(10,10)
      DO 10 J=0,90,10
      DO 10 I=1,10
   10 SUM=SUM + A(I+J)
```

Of course, since most FORTRAN compilers do not permit DO-loops to start at 0, the programmer will usually have to write his own loop control statements.

To measure the possible savings this type of change can achieve, I modified a CDC 6600 program in this way. Five minutes' effort resulted in a time savings of 3%.

Another comment regarding subscripted variables is in order. Frequently, several references to the same variable will be made. Instead of making all these references to the subscripted variable, it is more efficient to save the value of a variable in a simple variable and refer thereafter to that variable:

```
X=A(I,J)
.
. (statements using X)
.
```

III. LOOPS

In the great majority of programs, the bulk of the processing is performed within loops—usually within deeply nested loops. The efficient programmer will, therefore, hold the processing in his loops to a minimum and do whatever is really loop-invariant outside of the loop. While this sounds like a fairly simple idea, it is far too frequently violated.

One important example has already been given: the references to subscripted variables. It is very important to understand *how* address calculations were made loop-invariant. The original example had the I-loop outside and the J-loop inside (the way most programmers would tend to write). But this way, the calculation JJ=10*(J-1) is *not* loop-invariant. The trick is to reverse the nesting order of the two loops. Once the I-loop is inside, the JJ calculation can be removed from the inner loop.

This may be stated as a general principle: If arrays are *stored in row-major order* (that is, the row subscript varies most rapidly), then nest DO-loops so that the loop of the row subscript is innermost. If *column-major order* is employed by the compiler, then nest DO-loops with the loop of the column subscript innermost. This happens to be a compiler-dependent factor, but program efficiency is usually purchased at the cost of machine or compiler dependency.

IV. ARITHMETIC CALCULATIONS

The statements a programmer tends to write most automatically are those which perform arithmetic calculations and assignments.

Yet here, too, considerable savings in speed are frequently possible, especially if the arithmetic is performed within a loop. There are two basic points to keep in mind.

First of all, the arithmetic operations are not usually equally fast. On most machines, floating point arithmetic is slower than integer arithmetic; within each class, division is generally the slowest operation, then multiplication, and then addition and subtraction. Powers, roots, logarithms, and trigonometric operations are all implemented by subroutines. But these rules do have exceptions. For example, the CDC 6600 has only floating point multiplication. To multiply two integers, the machine must first convert them to floating point, then multiply them and convert the result back to integer format. This is clearly much more expensive than a simple floating point multiplication.

The efficient programmer, having learned the relative costs of the arithmetic operations on his machine, will be able to arrange his computations to favor the cheapest operations. Some examples follow.

Expressions involving small (integer) constants can usually be modified, so as to replace expensive operations with cheaper ones. Thus, the expression 2.0*A may be written as A+A, and A**3 can be replaced with A*A*A. It is true that some compilers will do the latter substitution on their own, but the programmer can make sure by doing it himself.

Another example involves loops. Consider the following normalization loop:

```
      DO 1 I=1,10
    1 X(I)=Y(I)/Z
```

This requires ten divisions; we can get by with one division and ten multiplications:

```
      Z1=1.0/Z
      DO 2 I=1,10
    2 X(I)=Y(I)*Z1
```

A final example involves algebraic simplification that the programmer can perform. The statement

```
A = (B+1.0/B)*(C+1.0/C)*(D+1.0/D)
```

requires three divisions, two multiplications, and three additions. But it is equivalent to

```
A = (B*B+1.0)*(C*C+1.0)*(D*D+1.0)/(B*C*D),
```

which requires one division, seven multiplications, and three additions. Assume that a division takes as long as three multiplications. Then

the first approach requires the equivalent of eleven multiplications and three additions, while the second requires ten multiplications and three additions. We can thus save one multiplication out of eleven, or 9%. A more precise estimate is obtained if we take the addition into account. Suppose addition is twice as fast as multiplication. Then the first approach requires the equivalent of 25 additions, and the second approach needs 23. The saving is then 8%. It is worth mentioning that the estimates given approximate the true figures for the 6600.

The second basic point with regard to possible savings in arithmetic operations concerns partial results. If a computation requires some intermediate result for two subsequent operations, it is usually worthwhile to first compute the intermediate result, assign the value to a simple variable, and then use that variable later:

```
X = SQRT(B*B - 4.0*A*C)
Y = (-B+X)/(A+A)
Z = (-B-X)/(A+A)
```

Sometimes it costs more to store and fetch a result than to compute it twice. In the above example, this was assumed to be the case for the value A+A (or 2.0*A). But the value assigned to X is clearly not in this exceptional category.

V. ARGUMENT PASSAGE AND "COMMON"

Whenever arguments or results are passed between a program and a subroutine or function via a calling sequence, overhead will accrue. This is because either the value or the address of each argument must be fetched and stored to a specified location or register. The overhead is therefore proportional to the number of elements in the calling sequence.

All of this overhead can be avoided by placing these elements into a COMMON block (in both the calling program and called subroutine, of course). Then the subroutine knows at its compile time where its arguments are to be found, and where it should place its results. None of this information need be passed during execution.

VI. BRANCHING

Executing a *branch* instruction on any computer requires time, whereas allowing control to proceed sequentially does not. The efficient programmer will take advantage of this fact to minimize the amount

of branching his program will have to execute. Consider the following schematic example:

```
      IF (Condition) GO TO 1
      .
      .
      .
      GO TO 2
    1 CONTINUE
      .
      .
      .
    2 CONTINUE
```

In this case, whether the condition is true or false, one branch instruction must be executed. But suppose the probability that the condition will occur is only 1%. We can arrange for control to proceed sequentially when the condition is false, at the price of two branch instructions whenever the condition is true:

```
      IF (Condition) GO TO 1
      .
      .
      .
    2 CONTINUE
      .
      .
      .
      RETURN
    1 CONTINUE
      .
      .
      .
      GO TO 2
```

Notice that the statements to be executed if the condition holds are placed after some unconditional branch point in the program.

There is a second useful technique in branching, and this is *table lookup*. The FORTRAN equivalent is the computed GO-TO statement. When a programmer must implement a multiple branch (that is, test for a number of conditions and perform different actions for each), he can frequently arrange for a simple integer-valued function of the conditions, and use the function value as the index of the com-

puted GO–TO. Suppose we wish to test if a variable X is positive, negative, or zero. Then:

```
N = ISIGN(X)+2
GO TO (1,2,3),N
```

If X is negative, N will be 1; if X is zero, N will be 2; if X is positive, N will be 3.

VII. SPECIAL EFFECTS

Many FORTRAN compilers (and those of other higher level languages) offer special, nonstandard features in their dialects. These features provide convenience, and in many cases, efficiency as well. I will cite some of the features that the CDC 6600 IDASYS FORTRAN provides.

(*a*) Several assignments may be specified in a single statement:

```
A=B=C=D=1.0/X
```

This is preferred to the sequence:

```
A=1.0/X
B=A
C=A
D=A
```

because it avoids the three extra "fetch" operations the letter sequence is likely to produce.

(*b*) A subroutine called ERASER will set all elements of an array to a specified value:

```
CALL ERASER (A,100,3.14)
```

will set the first 100 elements of the array A to the value 3.14. This is preferred to the DO-loop:

```
DO 1 I=100
1 A(I)=3.14
```

for the same reasons as in (*a*).

(*c*) Certain functions, based on some of the 6600's instructions, are compiled as in-line code; this precludes the overhead due to subroutine linkage. Examples are:

ISHIFT	:	shifts a word left (circular) or right.
LVAL	:	extracts a bit from a word.
MOD2	:	adds two words mod-2 bit-by-bit.
IDENS	:	counts the number of 1's in a word.
ISGN	:	returns a $-1,0,+1$ if the argument is negative, zero, positive, respectively.
SGN	:	similar to ISGN, but returns a floating point value.

VIII. PROGRAM SIZE

The type of efficiency we have been stressing is execution-time oriented. For many "third-generation" operating systems, this is only one factor. The amount of memory used by a program is equally important for these current systems. In fact, the accounting routines of some of these systems reflect this situation. Instead of charging the user only for his processor time, the system charges for memory usage and processor time. One approach is to multiply these two factors together and charge for the total amount of "word-seconds" used.

In such an environment, the user should seek to minimize his space-time product. Sometimes speed can be gained without a penalty in space; this is certainly a worthwhile change. Similarly, space can often be gained without a speed decrease. Judicious use of the FORTRAN EQUIVALENCE statement is one example of this process.

Most often, however, we face a tradeoff between speed and memory usage; and the decision is a harder one. For example, we might speed up a procedure by precalculating some large tables; during the procedure, we refer to the tables instead of repeating the calculations. In such a case, one should estimate the space penalty and the speed bonus, and determine which factor outweighs the other.

A final comment is in order regarding those multi-programming systems which still orient their accounting exclusively toward processing time. The user here will not be charged for wasted space, yet the system itself will suffer degraded performance, and the user will ultimately feel this in slower service. Hence, it is still to a user's advantage to reduce the memory requirements of his program if he can. In addition, to enforce this tendency toward smaller programs, system managers would be wise to amend their accounting procedures.

IX ALGORITHMIC EFFICIENCY

My next comments apply not to programming per se, but to the writing of algorithms. An algorithm is a detailed specification of the steps that must be performed to transform a given input to the desired output. Programs are thus examples of algorithms. Thus far, we've analyzed ways of writing more efficient programs, under the tacit assumption that the algorithm had already been produced. Clearly, one must also do his best to ensure that the algorithm he defines is efficient as well.

Frequently, one will write a bad algorithm because of an inadvisable data representation. Consider the following task: produce an algorithm to process alphabetic text and perform a frequency count; the text is 1000 characters long and is punched 50 characters to a card. One might approach this problem by defining a 26-long array of alphabetic con-

stants, and then comparing each character of text against these constants:

```
      INTEGER KONST(26), TEXT(1000), KOUNT(27)
      DATA KONST/1HA 1HB 1HC. . . .1HZ/
      READ (5,101) TEXT
  101 FORMAT (50A1)
      DO 60 I=1,1000
      J=TEXT(I)
      DO 40 K=1,26
   40 IF (J .EQ. KONST(K)) GO TO 50
      KOUNT(27) = KOUNT(27)+1
      GO TO 60
C     KOUNT(27) COUNTS GARBLES
   50 KOUNT(K) = KOUNT(K)+1
   60 CONTINUE
```

If the text is flat random, we expect thirteen passes through the inner loop for each character of text.

We can do much better than this by translating the binary-coded decimal (BCD) values stored in TEXT into integers (running from 1 to 26). This can be done rapidly via a translation table:

```
      INTEGER TRANTB(64)
      DATA TRANTB/. . ./
```

where

```
      TRANTB(I) = J
```

if the BCD value of the J-th character ($1 \leqq J \leqq 26$) considered as an octal number is I. In other words, if the BCD of A is 21_8, set TRANTB (17)=A, since $21_8 = 17_{10}$. Now we must read the text in R1 format to keep the characters right-adjusted:

```
      READ (5,101) TEXT
  101 FORMAT (50R1)
      DO 50 I=1,1000
      L = TEXT(I) +1
      J = TRANTB(L)
   50 KOUNT(J) = KOUNT(J)+1
```

As long as we fill the 64-26=38 values of TRANTB that don't correspond to alphabetic characters with the value of 27, this method is exactly equivalent to the first.

```
      INTEGER COUNT(27)
      READ(5,101) TEXT
      DIMENSION KOUNT(64)
  101 FORMAT (50R1)
      DO 50 I=1,1000
      J = TEXT(I) +1
   50 KOUNT(J) = KOUNT(J) +1
      DO 60 I=1,64
      J = TRANTB(I)
   60 COUNT(J) = COUNT(J) + KOUNT(I)
      C  ASSUMES KOUNT AND COUNT ARE ZEROIZED
```

The moral of this example is: Don't be restricted by conventional data representations, but search for the most convenient representation for your purposes.

A second example of this principle (from Professor T.E. Cheatham) will be described, but left as an exercise to the reader:

(1) Write an algorithm to put in two integers, divide the first by the second, and put out the quotient and remainder. You may use addition, subtraction, and multiplication, but not division (since, in effect, this is what you are defining). Assume that all numbers are in a decimal representation.

(2) Do the tasks of (1), but assume a Roman numeral representation.

X. CONCLUSION

The foregoing has been an attempt to motivate programmers to adopt more efficient techniques and to explain a number of possibilities. Two important caveats are in order.

First, not all of the suggestions are always going to work. For each technique there will be situations in which it will prove counter-productive. The programmer must analyze the applicability of these suggestions to his specific problem and system.

One method that may be used is controlled experimentation. The programmer faced with a choice of two methods can time them on his target system and then make a choice based on fact rather than belief.

As an example, consider the following statements (see above, section IV):

```
X = (A+1.0/A)*(B+1.0/B)*(C+1.0/C)
X = (A*A+1.0)*(B*B+1.0)*(C*C+1.0)/(A*B*C)
```

We can write a toy program, embedding each statement in a loop, so that it is iterated one million times. We can time each loop, and establish the percent difference between the methods. This experiment was performed on the 6600, and the results, even though the second was expected to be 8% faster, indicated that the two methods were equivalent. The explanation lies in the ability of the 6600 to execute some arithmetic operations in parallel. The sequences of instructions produced by the compiler for each statement determined the degree of overlap, and the first method was favored more. This example thus demonstrated not only the method of analysis that the programmer can use, but also the need for such analysis.

It must also be borne in mind that this is but a partial listing of efficiency techniques. It is intended more to stimulate additional thought on the subject than to serve as a handbook. Given the Agency's investment in computing power, it is a topic of no small interest and importance.