1984

Fortran Extensions for Parallel and Vector Computation

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
84-470

PURDUE CENTER FOR PARALLEL AND VECTOR COMPUTING
REPORT #9

FORTRAN EXTENSIONS FOR PARALLEL AND VECTOR COMPUTATION

John R. Rice
23 January, 1984
CSD-TR 470

ABSTRACT

This is an essay at defining a "minimal" extensions of Fortran which provides facilities for expressing parallel and vector algorithms well. Thus facilities are implemented by using Fortran style and syntax whenever possible. It is intended to be implemented as a Fortran preprocessor. Orthogonal facilities are proposed for computational control, defining and operating on a variety of data structures, and critical variables in shared memories.

FORTRAN EXTENSIONS FOR PARALLEL AND VECTOR COMPUTATION

John R. Rice

The object of this note is to explore the possibility of a "minimal effect" extension of Fortran that will be good (not just tolerable) for expressing parallel computations. One needs four faclities in this extension: *control statements* to express parallelism, *declaration statements* to define organized structures, *protection of critical data* in shared memories, and *computational statements*. These facilities are to be "orthogonal" in the following senses.

1. The control statements apply to any and all data structures.
2. Each data structure is put into the language processor independently with the only "generic" properties such as Size (storage allocation specifications), Range (working size) and Mask (logical valued image of the data structure - might be an expression involving index quantities). All existing Fortran operators may be extended "generically" to a data structure, otherwise operators on data structures are functions or subroutines. Functions may have data structure values.
3. Any Fortran variable (including data structures) may be declared CRITICAL and its access protected by intrinsic functions.

Not described here are a number of simple and non-controversial mechanisms e.g. a:b:c for "from a to b with stride c", A(9:100) for a segment of an array, A(*) for the entire range of an array. These mechanisms are to be patterned after the Fortran 8X proposals.

Our philosophy is to use existing Fortran syntax whenever possible so as to make the resulting language be Fortran-like, and hence familiar to engineers and scientists used to Fortran. The *compound statement* construct is useful in the parallelization of computations, but is not natural in Fortran and we have proposed nothing for it. Control statements are statement oriented (as in Fortran) and thus compound statements (or groups of statements) are needed to make this smooth. These are, of course, awkward constructs in Fortran to create groups of statements as in

```
IF(.TRUE.)THEN                DO 10 I = 1,1
      statements                    statements
ENDIF                         10 CONTINUE
```

A. CONTROL STATEMENTS

The data structures compatible for parallel and vector computing have one or more "regular dimensions" that are indexed naturally. The ranges of these indices are the natural control variable for control statements.

We propose the following two control statements:

#1 FOR ALL (Range specs, Mask) Computation statement

This provides for the systematic processing of all elements of a data structure without regard to sequencing. The results of this statement are to be the same as if it were done as follows

1. All variable values copied into temporary memory for each statement.
2. Computations made by each statement.
3. Resulting variable values copied back to permanent memory.

Thus in

```
FOR ALL (I = 2:9)    X(I) = (X(I-1)+2*X(I)+X(I+1))/4
```

all right side values are computed, and then assigned to the left side variables.

Examples (syntax is for illustration purposes)

```
FOR ALL (I = 1:100, J = -10:10:2; I.NE.J) A(I,J) = (I+J)**2/SIN(I-J)
FOR ALL (I = 1:100; .NOT. GOING(I)) CALL RESTART(X,Y,I)
FOR ALL (I = 1:100)
        A(I,I) = I**2+1
        FORALL (J = 1:100; I.NE.J)
                A(I,J) = I*J + COS(I,J)
        END ALL
        NORM(I) = SUM(A(1:I-1,2), 1:I-1)
END ALL
```

Statements within a FORALL - END ALL are executed in strict sequential order, but the components of each statements have no implied ordering.

#2 DO PARALLEL

END PARALLEL

The unsystematic processing of a set of statements without regard to synchronization or sequencing.

Examples

```
DO PARALLEL
        CALL START1 (LEFT,SIDE)
        CALL START1 (RIGHT,SIDE)
        CALL SETUP (MIDDLE,BLOCK)
END PARALLEL
DO PARALLEL (ROW = 1:NROW)
        FOR ALL (COL = 1:NCOL)
                UNEW (ROW,COL) = RELAX(A,U,ROW,COL)
            CHANGE(ROW,COL) = ABS(UNEW(ROW,COL) - U(ROW,COL))
        END ALL
END PARALLEL
```

Note that the following are quite different in effect

```
DO  PARALLEL                    FOR ALL (I = 1:9)
   X(I) = X(I+1)+X(I)             X(I) = X(I+1)+X(I)
END PARALLEL                    END ALL
```

The indexing of a DO PARALLEL simply replicates the statements within its scope with the indicated indexes, no ordering within these statements is implied.

One can consider replacing the DO PARALLEL construct by a BARRIER construct (essentially a join and fork). Indeed, the END PARALLEL is just a BARRIER statement and the DO PARALLEL is often redundent. Arguments for the BARRIER include:

1. Less verbose than DO PARALLEL
2. Data flow analysis can identify the "highest" DO PARALLEL for a given END PARALLEL
3. It is simpler and just as powerful

Arguments for the DO PARALLEL include

1. The scope of parallelism is clearly displayed.
2. Side effects of statements are automatically considered by asking the programmer to specify the scope of parallelism. Data flow analysis does not handle side effects properly.

3. It is simple and just as powerful
4. It allows replication naturally.

It is clear that the FOR ALL and DO PARALLEL allow general parallel operations to be expressed. A key question is the following. Suppose one has *M* processes to do and knows that these should be divided into *K* sets for maximum efficiency. One might hope that the language translator would do this properly and automatically, but this is rather optimistic. How do these constructs allow this? Three simple mechanisms are illustrated:

```
      DO 10 L = 1,K
            FORALL(2 = I:M-K+L:K) A(I,L) = A(I,L)+A(I+1,L)+F(I,L)
10    CONTINUE

      DO PARALLEL (L = 1:K)
            DO PARALLEL (I = L:M-K+L:K)
                  CALL PROCESS(I)
            END PARALLEL
      END PARALLEL

      DO 10 L = 1,K
            DO PARALLEL(I = L:M-K+L:K)
                  CALL PROCESS(I)
            END PARALLEL
10    CONTINUE
```

B. DATA STRUCTURES

We propose an unlimited (in principle) number of data type declarations. The plan is to have them independent of one another and to provide mechanisms for indexing that do not depend on the specific data type. Each data type requires considerable specific support internal-to-the-language-processor-and-we-do-not-propose-an-abstract-data-type-facility. Each data type has four associated "variables" (these can be fairly complex in practice)

DESCRIPTOR: The values of all variables that define the data structure.

SIZE: The size of the storage allocated for the structure. This is expressed in terms of the parameters in the descriptor.

RANGE: The current size of the structure.

MASK: A logical structure that matches the original structure and which determines elements to be included or excluded in processing.

It is visualized that the data structures have a substantial "systematic" nature and that there are indexes which "range over" the structures. There is a set of data type specific functions to access detailed information about the size and range of a given structure. Operators on new data types are either the standard Fortran operators (+, -, /, *, .LE., .AND., ...) or functions or subroutines.

Example 1. ARRAYS

Descriptor = Dimension
For K=1 to Dimension
lower bound≤ lower range≤ upper range≤ upper bound

stride

Declaration:
DIMENSION A(LB:UB, LB:UB,...)
Size/Range: Lower, Upper, Stride, ..., Lower, Upper, Stride

Example 2. MATRICES
Matrices are handled like two dimensional arrays with the declaration MATRIX. Some operators and functions are different.

Example 3. SPARSE VECTOR
Descriptor = lower bound≤ lower range≤ upper range≤ upper bound
stride
name (vector or function to produce included elements)
max_length
Declaration:
SPARSE VECTOR X(max_length)
Size/Range Lower, Upper, Stride
Index: Name, max_length

Example 4. TREE (with indices depth and index≤ b^{depth})
Descriptor: Branching Factor $b \geq 2$
depth range≤ depth bound
Declaration:
TREE TRITREE(3,10)
Size/Range: depth bound, depth range

Example 5. DOUBLE STACK
Descriptor: max_length
(optional) top-bound≥ top-range≥ bottom-range≥ bottom-bound
Declaration:
STACK STACK1(1000,1,*)
STACK QG(1000,*,*)
Range/Size: bottom, top

The elements of these structures can be any simple variable type: REAL, INTEGER, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, DOUBLE PRECISION COMPLEX.

C. PASSING DATA STRUCTURES TO/FROM PROCEDURES

We assume that any of the data structures included in the extension can be an argument or value of a procedure. The descriptor information is automatically made available by the system through inquiry functions. There is a stack for the block allocation of storage for structures. Arguments are passed by reference when practical.

Example 1: Subroutine with matrix arguments

```
      SUBROUTINE EXAMP1(DS1, DS2, VAR1, VAR2, MASK1, VAR3)
C                        INPUT DECLARATIONS
      BAND MATRIX DS1(*)                          Band Matrix
      DIMENSION DS2(*,*)                          2-D Array
      LOGICAL MASK(*,*)                           2-D Mask
      BAND MATRIX RANGE VAR1                      Range vector
```

```
      ARRAY RANGE VAR3                                  Range vector
      INTEGER VAR2                                      Simple variable
C                          LOCAL DECLARATIONS
      BAND MATRIX TEMP(DESCRIPTOR(DS1))                 Structure of DS1
      INTEGER ARRAY INDICES(DESCRIPTOR(DS2))            Structure of DS2
      LOGICAL VECTOR ROWCOPY(LBOUND(MASK,1):UBOUND(MASK1))
                                          Structure derived from MASK
```

Example 2: Function for $f(j,k) = \sum_{i=1}^{N} a_i \phi_i(x_j)\psi_i(y_k)$

This example uses the inquiry functions RANGE, URANGE and LRANGE to obtain the current size of vectors. A special SUM function is used for summing along the 3rd dimension of an array, this is in the proposed Fortran 8X, but is not part of the extension disussed here.

```
      FUNCTION F(A, PHI, PSI, X, Y)
      DIMENSION  A(*), X(*), Y(*)
      DIMENSION F(LRANGE(X):URANGE(X),LRANGE(Y):URANGE(Y))
DIMENSION TEMP(LRANGE(X):URANGE(X),LRANGE(Y):URANGE(Y),LRANGE(A):URANGE(A))
      N1 = LRANGE(A)
      N2 = URANGE(A)
      FOR ALL (J = LRANGE(X):URANGE(X),K = LRANGE(Y):URANGE(Y))
            TEMP(J,K,*) = A(*)*PHI(*,X(J)*PSI(*,Y(K))
            F(J,K) = SUM (TEMP, INDEX=3, RANGE = N1:N2)
      END ALL
      END
```

The FOR ALL here can be replaced by DO PARALLEL. An alternative body of the function is:

```
      DO PARALLEL (J = RANGE(X), K = RANGE(Y))
            F(J,K) = 0
            DO I = RANGE(A)
                  F(J,K) = F(J,K) + A(I) * PHI(I,X(J)) * PSI(I,Y(K))
            END DO
      END PARALLEL
```

An alternative which is essentially different is

```
      FUNCTION          F(A, PHI, PSI, X, Y)
      PARAMETER         (LX = LRANGE(X), LY = LRANGE(Y), LA = LRANGE(A) &
                        UX = URANGE(X), UY = URANGE(Y), UA = URANGE(A))
      DIMENSION PHIVAL(LA:UA,LX:UX), PSIVAL(LA,UA,LY:UY)
      INTEGER LX,LY,LA,UX,UY,UA
      DO PARALLEL
         DO PARALLEL(I = LA:UA, J = LX:UX) PHIVAL(I,J) = PHI(I,X(J))
         DO PARALLEL(I = LA:UA, K = LY:UY) PSIVAL(I,K) = PSI(I,Y(K))
         DO PARALLEL(J = LX:UX, K = LY:UY) F(J,K) = 0.0
      END PARALLEL
      DO PARALLEL (J = LX:UX, K = LY:UY)
```

```
        DO I = UA:LA
              F(J,K) = F(J,K) + A(I)*PHIVAL(I,J)*PSIVAL(I,K)
        END DO
END PARALLEL
END
```

D. CRITICAL VARIABLES (SHARED MEMORY ACCESS)

To protect critical values in shared memory we propose that they first be declared critical and then accessed by normal Fortran mechanisms. The critical value access functions ACCESS, RELEASE, GET and PUT are used as ordinary Fortran functions.

Example 1: Modify a critical vector

```
SUBROUTINE UPMAX (NUVALU)
DIMENSION NUVALU(*)
CRITICAL COMMON / MAXVALU / SACRED(1000)
CALL ACCESS(SACRED)
        FORALL (I = 1: URANGE(NUVALU)) SACRED(1) = AMAX1(SACRED(I),NUVALU(I))
CALL RELEASE(SACRED)
RETURN
END
```

The example illustrates the basic facilities, ACCESS and RELEASE, that are part of the Fortran extension system. A shorthand version of this facility will be available for simple read/writes as in

```
CALL GET(SACRED,TEMP)          Copy value from SACRED
CALL PUT(SACRED,NUDOGMA)       Copy value to SACRED
```

E. IMPLEMENTATION NOTES

The context for the implementation is:

1. Want to be able to target several machines, e.g. Cyper 205, Cray, HEP, GF10,...
2. Want to allow flexibility for experimentations with the source language.
3. Want to minimize effort of implementation.
4. Translation speed is secondary unless it's terrible.
5. Want to have a Fortran based, portable implementation.

We list several choices along with their strength and weakness. Note that the preprocessor will have to do, at least, a lexical analysis of every statement in the program.

Choice #1. Direct Fortran

Strengths:	Simple concept
	Not dependent on anyone else
	Easy installation
Weaknesses:	More work
	Less flexibility

Choice #2. PG System and TOOLPACK Template Processor

Strengths:	Nice grammar approach
	Flexibility is very good
	Minimal effort
Weakness:	Must have PG expertise
	Must maintain PG system
	Templates use lots of memory and I/O

Choice #3. Protran System of IMSL for Creating Preprocessors

Strengths	Good "grammar" approach
	Flexibility is very good
	IMSL maintains system
Weakness:	Must have PROTRAN System expertise

Choice #4. Build on Kuck's PARAFRASE System

A schematic of this system is given

Strengths:	PARAFRASE does the machine dependent code generation
	Much less work
Weaknesses:	Must have PARAFRASE expertise
	Must have access to PARAFRASE

Conclusion: Choice #4 is the best except for the PARAFRASE access problem. Choices #2 and #3 are comparable and choice #1 is the worst.