Parallel Programming Language ADETRAN

By

Tatsuo NOGI

(Received June 30, 1989)

Summary. The programming language called ADETRAN is described. It was developed for the parallel computer ADENA on the basis of FORTRAN 77. It is designed powerful enough to support some parallel control commands. In view of its FORTRAN-likeness, emphasis was placed on parallel syntax and control.

Contents

1. Introduction ………… 236
2. Concepts of ADENA computer and ADETRAN language ………… 237
2.1 Simulation scheme ………… 237
2.2 Data structure and parallel control in ADETRAN ………… 241
3. Elements of the language ………… 242
3.1 Characters ………… 242
3.2 Name ………… 242
3.3 Value ………… 242
3.4 Key words ………… 243
3.5 Formatting ………… 243
4. Subprogram ………… 243
4.1 HOST subprogram ………… 243
4.2 SLAVE subprogram ………… 243
5. Data ………… 244
5.1 Constant ………… 245
5.2 Variable ………… 245
5.3 Variable array ………… 246
5.4 Record array and file array ………… 250
5.5 File-processing statements ………… 252
6. Common data area ………… 253
6.1 Common block ………… 253
6.2 Equivalenced storage area ………… 254
7. SLAVE system specification ………… 255
7.1 Parallel index region ………… 256
8. Expressions ………… 256
8.1 Kinds of expressions ………… 257
8.2 HOST and SLAVE expressions ………… 258
9. Assignment statement ………… 258
10. General control statements ………… 259
10.1 GO TO statement ………… 259
10.2 IF statements ………… 259
10.3 Repetitive statement ………… 262

Department of Applied Systems Science

10.4 General punctuation statements 263
11. Parallel control statements 264
11.1 PDO statement 264
11.2 Parallel accumulation operation 266
11.3 PASS statement 266
11.4 PHASE statement 268
11.5 POUR statement 270
11.6 PAD statement 272
11.7 PUSH statement 273
12. SLAVE system synchronizing statements 274
12.1 Logical IFALL/IFANY statement 274
12.2 Block IFALL/IFANY statement and ENDIFA statement 276
12.3 ELSEIFALL/ELSEIFANY statement 276
12.4 ELSEA statement in IFALL/IFANY block 277
13. FUNCTION 277
13.1 Statement function statement 278
13.2 FUNCTION and LFUNCTION subprogram 279
14. SUBROUTINE and LSUBROUTINE subprogram 280
15. Global subprogram 282
15.1 GSUBROUTINE subprogram 282
15.2 GSUBCOROUTINE subprogram 283
16. CALL statement 283
17. Other subprogram related items 285
17.1 RETURN statement 285
17.2 Actual argument 285
17.3 EXTERNAL statement 286
17.4 INTRINSIC statement 286
17.5 Dummy argument 287
18. Othen HOST statements 287
18.1 PROGRM statement 288
18.2 ENTRY statement 288
18.3 Input/Output statements 288
18.4 Data initialization 289

1. Introduction

The ADETRAN is the new high-level parallel language tailored for the parallel machine ADENA, which was designed especially for solutions of simulation models in science and engineering. The ADETRAN language is an extension of the FORTRAN 77 language, and it contains, in addition to the full FORTRAN, ADETRAN PROPER having some parallel control commands. The important characteristics of the ADETRAN are seen in its array data structure, which directly reflects the scheme of being shared among processors. The structure then allows explicit indication of parallel computations, and expresses well the machine architecture.

The ADETRAN language was designed so to be opened that the machine architecture may be grasped as far as necessary by users, and hence, parallel

programs may be written easily by users themselves. Though this approach is different from that taken for vector pipeline computers, which supply the automatic vectorizer that produces vector-oriented object codes from FORTRAN source programs, we think that our approach is better since an efficient solution can be, in general, obtained by beginning with designs of models or computational schemes and algorithms. It is then important that users themselves can design those models or algorithms with the prospect of parallel processing. To have such a prospect, language is essential, and we consider that the ADETRAN is powerful enough to give users the useful image of the ADENA.

The paper. gives the outline of ADETRAN. Chapter 2 gives the concepts of the ADENA computer and the ADETRAN language. All chapters beginning with Chapter 3 give the ADETRAN syntax and semantics, and tell its usage.

Acknowledgement The design of the ADETRAN and its compiler implementation has been supported by many persons, especially by Mr. A. Wakatani, Mr. T. Okamoto, Mr. J. Nishikawa (Matsushita Electric Industrial Company) and Mr. K. Sugiyama (IBM), and by all staffs of the Semiconductor Research Center of Matsushita Electric Industrial Company which has been engaged in the development of the ADENA Computer (the chief leader being Mr. H. Kadota). The author deeply thanks them. He also thanks Dr. H. Mizuno and Dr. S. Horiuchi (Matsushita Electric Industrial Company) for their encouragement. He also thanks Miss N. Maruyama for typewriting his manuscript.

2. Concepts of ADENA computer and ADETRAN language

2.1 Simulation scheme

2.1.1 Simulation models in science and engineering are broadly divided into two classes: continuum and particle models. A continuum model is usually given by a system of partial differential equations (PDEs), while a particle model is given by a set of kinetic equations of all particles. They are often mixed in some complex problems.

Continuum models produce, through discretization, some natural 1-, 2-, or 3-dimensional variable array data to be solved or to be augumented, whose subscript indexes correspond to mesh point numbers counted along the space coordinate axes, respectively. Those variable array data are the very fundamental data structure in simulation.

Particles models demands record/file array data. That is, each element of the array is a file, each ingredient of which is a record that contains information of each particle.

Thus, we consider the variable and file array as the basic data structure in simulation.

2.1.2 Parallel computation is, in general, performed through independent computations by many processors and data transfer among those processors. The chief point in independent computations is seen in the way of sharing array data among processors. The most popular is to assign each processor to each number coordinate of the array (i. e. to each mesh point). We call such way to share data the *pointwise sharing scheme.*

The ADENA computer takes the other way. It is to assign each processor to each number coordinate of the array (i. e. to each sequence of mesh points on a line segment parallel to a coordinate axis). We call this way the *segmentwise sharing scheme.*

A difference between the most popular scheme and the ADENA's is seen in whether fully parallel or partially parallel and serial can be expected in the first place. The latter aims to process each array segment (which is produced with one or two fixed coordinate numbers remaining a free coordinate number) in serial.

The difference may, however, disappear in considering that it is impossible to supply so many processors to cover all mesh points. Hence, it is obliged to assign each processor to each block composed of some neighbouring mesh points for which each processor is demanded to process in serial.

It is essential which way we should select. This problem cannot be solved only by the way of sharing data, and the solution must be sought through considering the data transfer schemes and further applicable algorithms.

2.1.3 The chief problem of the data transfer schemes is to reduce overhead due to data transmission as far as possible. Such overhead determines the efficiency of the parallel computation. It may be true that the more connection paths among processors there are, the more such overhead is reduced. However, it is severely restricted by the difficulty of supplying so much hardware of connection passes.

The most popular is to supply real data paths so that processors arranged in the form of a sequential, square or cubic grid may pass data to each other between any two neighbouring processors, respectively. This scheme, combined with the pointwise sharing scheme, produces the simplest parallel computers (for example, ILLIAC–IV, DAP, or PAX etc.). Data transmission between any two neighbouring processors may be realized directly and in a short time, but it must be rememberd that data transmission would produce 'bare' overhead as it stands. Further, such transmission ability is too poor to cover many sophisticated

algorithms.

2.1.4 What are sophisticated algorithms? We should here return to computational models which arises in usual 2- or 3-dimensional simulations. In this respect, we have already prescribed data arrays. For the next step, we must consider their processing way. It may probably be said that the more sophisticated the algorithm becomes, the more the serial processing part (which can not be reduced to parallel processing) it contains, even though it seems paradoxical. That is to say, sophistication and parallelism may contradict each other.

To answer sophistication and parallelism, both of which aim for high speed computation itself, we had to find the solution of the contradiction considering construction or selection of simulation schemes or algorithms. The solution has been found in computational mathematics. It can be schematically formulated in that one-dimensional processing relies upon a serial sophisticated sequence of computation after separating the original problem to the other dimensional directions into independent one-dimensional subproblems. Each dimension should be selected alternatingly.

That is, such 1-dimensional processing is first performed to the x-axis, secondly to the y-axis, and thirdly to the z-axis, and its cycle is repeated. This scheme was first found in the solution of the 2-dimensional heat equation and is now called the *ADI method.* It and its successors are, at present, broadly applied and have become the main methods for multi-dimensional simulations, and have gained some generic names, *fractional step methods* or *splitting-up operator methods* etc..

Such methods are, on one hand, so much sophisticated as to retain the same computation load in order as the full sophisticated method. On the other hand, they have so much parallelism to cover one or more dimensions. In other words, one-dimensional processing may absorb many 'nutritive' elements of more sophisticated algorithm, and may happen to be an appropriate unit of parallel processing.

We arrived at a key word in the ADENA world. It is '*segment*' which means each one dimensional subarray with respective fixed subscript values of other dimensions, extracted from the original 2- or 3-dimensional arrays. With the word 'segment', the basic processing style is summarized as follows: firstly process the sets of segments with the x-direction, secondly those with the y-direction, and thirdly those with the z-direction, if any, and repeat such a cycle as many times as necessary.

The ADENA's data sharing scheme mentioned above can be hence called the

segmentwise sharing scheme, with all processor sharing respective segments, and processing them in parallel.

2.1.5 How do processors share those segments with different directions? The simple solution is that all the processors share all segments with a specific direction and process them at one time, and those with another direction at the next time, and so on.

Processing those segments with different directions cannot be performed independently in usual problems, and the results of the present process must be applied for the succeeding process. It means the necessity to edit the set of segments with a direction into that with another direction. We call such an edition the 'Alternating Direction Edition' or simply *ADE*, which is also a key word of ADENA.

In this way, we have decided to regard the basic data transmission as ADE, and to construct a network architecture suitable for ADE. In this regard, we call our machine the ADENA (Alternating Direction Edition Nexus Array) computer.

2.1.6 The network architecture is similar to the crossbar switch arranged in the form of a cubic grid, with each switch being replaced by a buffer memory of the First-In-First-Out type. We here omit the details of physical realization, and we shall mention only the logical structure effectively realized by the physical architecture.

The square array of processors is provided, and it may stand against the cubic grid of buffer memory units, to anyone of the x-, y-, and z-directions at one time. Standing to each direction, each processor can access a sequence of buffers on the corresponding grid line segment stretched to the direction. This scheme realizes ADE easily. Standing to a direction, say x-direction, all processors write their respective segment elements into corresponding buffers respectively, and then, stand to another direction, say y-direction. They read their respective segment elements from corresponding buffers respectively. Then it is considered that the ADE from the x-direction to the y-direction was completed.

The two-dimensional ADE also can be effectively realized by the above three-dimensional physical architecture. We here omit its details.

2.1.7 Finally, it should be mentioned that the ADE scheme may veil the data transmission overhead. In fact, for the computation followed by an ADE, each processor with the direct memory access ability for data transmission can process corresponding segment elements successively, and write the results into the buffers immediately after getting, element by element, in the concurrent way, and reads the contents in buffers from the other standing position. We call this

concurrent way the *S-scheme.* In this respect, it is essential that each processor is obliged to process segment elements successively. During such a sequence, an ADE can be almost completed. This is the reason that we have selected the segmentwise sharing scheme.

2.2 Data structure and parallel control in ADETRAN

2.2.1 A parallel computation is performed for the array segments with a specific direction. This aspect can be expressed for data with the direction attribute by the indication of the parallel processing. For illustration, we will give a simple example:

```
        PDO J= 1,16, K= 1,32
        DO  10 I= 1,64
    10  U(I,/J,K/)=V(I- 1,/J,K/)+V(I+1,/J,K/)
        PEND
```

The PDO statement opens the PDO paragraph (which contains the DO paragraph) and the PEND statement closes it. U(I, /J, K/) is the I th element of /J, K/−th segment which is shared by the /J, K/−th processor. Permissible segment data must have the same subscript variables in slashes in a PDO paragraph, whose range is given in the PDO statement as seen as 'J= 1,16, K= 1,32'. The above PDO paragraph is considered to process segments with the x-direction, indicated by the slashes' position. The segments with the y- and z-directions would be expressed as

U(I/, J, /K) and U(/I, J/,K)

respectively. Each subscript variable outside of the slashes indicates an element of the segment. Note that for the segment with the y-direction U(I/, J, /K), K and I are in slashes and J is outside of the slashes. In a PDO paragraph, any FORTRAN statement may appear, while processed data are segments with a specific direction. An ADE is realized only by a PASS paragraph as, for example,

```
PASS I= 1,64,  J= 1,16,  K= 1,32
U(/I,J/,K)=U(I,/J,K/)
PEND
```

which indicates the ADE from the x-segments to the z-segments. The PDO and PASS statements are two basic parallel processing control statements. In addition, there are some special parallel processing statements: PUSH, PHASE, POUR, and PAD.

2.2.2 There is another important concept of data besides the segment. It is the vector which has a special meaning in ADETRAN. It should be taken that a constant or variable has as many components as the physical processor elements, each component shared by a corresponding processor. Therefore, we had better call it the *constant* or *variable vector* strictly. Applying this concept to the array segments, we also call the data set of elements with a fixed subscript outside of the slashes in the array the *vector*. Its components are here shared by the logical processors.

This use of the word is reasonable, because those vectors themselves would be objectives of the *vector-pipeline processing*. In comparison, we may call processing by the ADENA the *segment-parallel processing*.

3. Elements of the language

3.1 Characters

The source program characters of ADETRAN follows those of FORTRAN. Their alphabetical, numerical and special characters are just the same as in FORTRAN.

3.2 Name

A name is used in a program unit to identify such an item as a constant, a variable, an array, a function, a subprogram and a common block, etc. Its use is just the same as in FORTRAN.

3.3 Value

Several kinds of values are allowed and are the same as in FORTRAN.

3.3.1 *A numerical value* is expressed as a decimal number, and it is simply called a number. It is anyone of integer, real and complex values.

3.3.2 *A logical value* specifies either true or false. They are expressed as .TRUE. and .FALSE. respectively.

3.3.3 *A character value* is a string of characters of alphabetical, numerical, and/or special characters, delimited as follows:

1. The string can be enclosed in apostrophes.
2. The string can be preceded by wH where w is the number of characters in the string.

Examples:

'ADENA-2' 7 HEXPO'90

3.4 Key words

Therse are predefined key words having special meanings. They are FORTRAN key words and added ADETRAN PROPER key words. They can not be used for other items.

3.4.1 FORTRAN key word : =

ASSIGN | BACKSPACE | BLOCKDATA | COMMON | COMPLEX | CONTINUE | DATA | DIMENSION | DO | DOUBLEPRECISION | ELSE | ELSEIF | ENDIF | ENTRY | EXTERNAL | FORMAT | FUNCTION | GOTO | IF | IMPLICIT | INQUIRE | INTRINSIC | LOGICAL | OPEN | PARAMETER | PAUSE | PROGRAM | READ | REAL | RETURN | STOP | SUBROUTINE | WRITE

3.4.2 ADETRAN PROPER key word : =

ELSEA | ELSEIFALL | ELSEIFANY | ENDIFA | FILE | GET | GSUBCOROUTINE | GSUBROUTINE | IFALL | IFANY | LFUNCTION | LSUBROUTINE | PAD | PASS | PDO | PEND | PHASE | POUR | PUSH | PUT | RECORD | REGION | RESET | REWRITE

3.4.3 ADETRAN-supplied mathematical function names also are considered as key words, and they also are the same as in FORTRAN.

3.5 Formatting

The statement of an ADETRAN source program can be written in a standard FORTRAN coding fixed form. Rules about 80-column card, comment line, statement number, continuation line and blank column are the same as in FORTRAN.

4. Subprogram

A program usually consists of several subprogram units. We simply call such a subprogram a program unit. There are two broad kinds of ADETRAN subprogram : the HOST and SLAVE subprogram.

4.1 HOST subprogram

A HOST subprogram is for the HOST computer, and it is nothing but a FORTRAN subprogram. There are four kinds of HOST subprograms : MAIN program, SUBROUTINE (subprogram), FUNCTION (subprogram) and BLOCKDATA (subprogram). An ADETRAN MAIN program must exist.

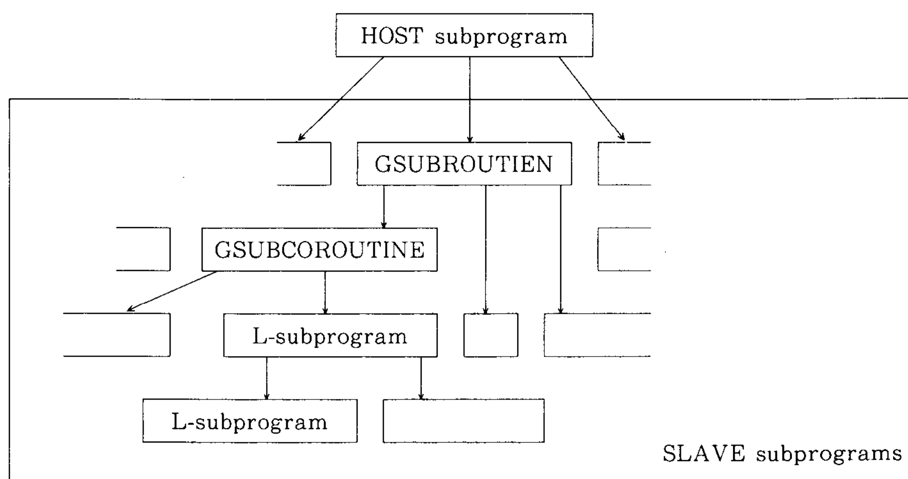
4.2 SLAVE subprogram

A SLAVE subprogram is for the SLAVE system. SLAVE subprograms are classified broadly into two categories: global and local subprograms.

4.2.1 *A global subprogram* is a parallel processing program for the whole SLAVE system. There are two kinds of global subprograms: GSUBROUTINE (subprogram), and GSUBCOROUTINE (subprogram). Any HOST subprogram (except a BLOCKDATA) can call a GSUBROUTINE and hence drive the SLAVE system. A GSUBROUTINE can not call any other GSUBROUTINE, much less a HOST subprogram. A GSUBROUTINE may call a GSUBCOROUTINE.

4.2.2 *A local subprogram* is a FORTRAN program for SLAVE data, common to all prosessing elements of the SLAVE system, and it is executed by being referred to in a global subprogram. There are two kinds of local subprograms: LSUBROUTINE (subprogram) and LFUNCTION (subprogram). Any global subprogram can call or quote a local subprogram. Any local subprogram can call or quote another local subprogram, but cannot call or quote a global subprogram or a HOST subprogram.

4.2.3 Reference relation among several kinds of subprograms are illustrated in the following diagram:



5. Data

Data are computation objects, and classified broadly into three categories: constant, variable, and array. Data also are divided broadly into two classes according to where to be stored, in the HOST computer or in the SLAVE

system : *HOST* and *SLAVE data.*

There are five basic kinds of data types : INTEGER, REAL (DOUBLEPRECISION), COMPLEX, LOGICAL and CHARACTER. CHARACTER data may only appear in a HOST subprogram. In a SLAVE subprogram, distinction between REAL and DOUBLE PRECISION is removed and REAL data are always taken as DOUBLEPRECISION.

5.1 Constant

5.1.1 A constant is a fixed, unvarying quantity. According to its value type, it is classified into an INTEGER, REAL (DOUBLE PRECISION), COMPLEX, LOGICAL, or CHARACTER constant.

5.1.2 *A HOST constant* is reserved in the HOST computer. It is *a constant scalar.* A HOST constant may appear in a HOST subprogram. *A SLAVE constant* is reserved in the SLAVE system. It is a *constant vector* whose components have the same value and are reserved in all physical SLAVE processors (strictly in their private storages) respectively. A SLAVE constant may appear in a SLAVE subprogram.

5.1.3 Any constant may be assinged to a name. Such specification is given at the head part of the subprogram by a PARAMETER statement (with the same format as in FORTRAN).

Example :

```
PARAMETER(TIMES=100, N=32, PAI=3.1415)
```

5.2 Variable

A variable is a variable data item, identified by a name. According to its value type, it is classified into an INTEGER, REAL, COMPLEX, LOGICAL or CHARACTER variable.

5.2.1 *A HOST variable* occupies a storage area in the HOST computer. It is *a variable scalar.* A HOST variable may appear in HOST subprogram. A SLAVE variable occupies each storage area of the same number as those in physical prosessors. It is *a variable vector* whose components are of the same number as the physical processors. Each component of a variable vector cannot be referred to individually, but the vector itself is referred to as an entity of data items.

5.2.1.1 A variable vector whose components all have the same value is especially called *a variable uni-vector.* It may appear, for example, as a 'DO variable' in a global subprogram.

5.2.2 The type of a variable may be declared by the following three

methods: except for the first method, a type specific name is used, which is one of the following: INTEGER, REAL, DOUBLEPRECISION, COMPLEX, LOGICAL, or CHARACTER [*len], *len specifies the length in bytes.

5.2.2.1 *The declaration by the predefined specification*

The predefined specification is a convention used to specify variables as INTEGER or REAL as follows:

1. If the first character of the variabel name is I, J, K, L, M, or N, the variable is INTEGER.
2. If the first character of the variable name is any other alphabetical character, the variable is REAL.

5.2.2.2 *Type declaration by the IMPLICIT statement*

The IMPLICIT statement may specify the type of variables, in much the same way as was specified by the predefined convention. That is, in both, the type is determined by the first character of the variable name. Its format is the same as in FORTRAN.

Example:

```
IMPLICIT INTEGER(H-K),   LOGICAL(L, M)
```

The IMPLICIT statement overrides the variable type as determined by the predefined convention. For example, under the IMPLICIT statement mentioned above, the variable name HI has the INTEGER type and LAST has the LOGICAL type.

5.2.2.3 *Type Declaration by explicit specification statements*

Explicit specification statements differ from the first two ways of specifying the type of a variable, in that an explicit specification statement declares the type of a particular variable by its *name* rather than as a group of variable names beginning with a particular *letter*. Its format is the same as in FORTRAN.

Examples:

```
INTEGER  COUNT
REAL     MESH
```

The explicit type statements override the IMPLICIT statement, which, in turn, overrides the predefined convention for the specifying type.

5.3 Variable array

5.3.1 A data array is set of variable items of the same type identified by a name, called the *variable array name*. The variable items which the variable array comprises are called *variable array elements*. A particular element in the

variable array is identified by the variable array name and its position in the variable array (e. g. first element, third element, etc.). The number and arrangement of elements in a variable array are specified by the variable array declarator. The variable array declarator indicates the number of dimensions, the size of each dimension and the direction attribute. To refer to any element in a variable array, the array name plus a parenthesized subscript must be used.

5.3.2 *A HOST variable array* occupies a continued block of storage area in the HOST computer. It is nothing but a usual FORTRAN array. A HOST variable array may appear in a HOST subprogram, and may have a maximum of seven dimensions. A HOST variable array may also appear in a GSUBROUTINE for the specific purpose of transferring data between the HOST computer and the SLAVE system.

Examples:

U(I), V(I, J), W(I, J, K)

represent specific elements of the HOST variable array U, V, and W, respectively.

5.3.3 *A SLAVE variable array* occupies corresponding memory blocks of all specified SLAVE processors. It is considered as a segment array.

5.3.4 *A segment array* may or may not have a direction attribute. A segment array with a direction attribute is called a *directed segment array.* A segment array without any direction attribute is called a *non-directed segment array.*

5.3.5 *A directed segment array* may have either 2 or 3 dimensions. A 2-dimensional directed segment array may have either an x- or y-direction. A 3-dimensional directed segment array may have one of an x-, y- or z-direction. Here we use the usual mathematical convention, x-, y-, and z-directions, which correspond to 1 st, 2 nd, and 3 rd subscript indices, respectively.

Direction attributes are expressed by the ways in which subscripts are enclosed by a pair of slashes.

Examples: Two 2-dimensional variable array elements

U(I, /J/) and U(/I/, J)

are those of the variable array U with an x-direction and the variable array U with a y-direction, respectively.

Three 3-dimensional variable array elements

V(I, /J, K/), V(I/, J, /K), and V(/I, J/, K)

are those of the variable array V with an x-direction, with a y-direction and

with a z-direction, respectively.

Even if some segment arrays with different direction attributes have a same name, they are different data items. Hence, it is better to consider that each directed segment array is named together with its direction attribute.

5.3.5.1 In a directed segment, a subscipt index (es) enclosed by a pair of slashes is called *the processor number* (*index*). The element of the segment array with the x-direction V(I, /J, K/) seen above has the processor number index /J, K/. The element of segment array with the y-direction V(I/, J, /K) has the processor number index /K, I/, where it is considered that K and I are enclosed in that ordered by the pair of slashes.

For a specific processor number, we always have a one dimensional subarray whose components are specified by the subscript index outside of the pair of slashes, which we call a *segment*.

Each segment of a segment array with an x-, y-, or z-direction is called an x-direction segment, a y-direction segment or a z-direction segment alternatively. These segments are totally called *directed segments*.

5.3.5.2 Directed segments may appear only in a global subprogram.

5.3.6 *A non-directed segment* array has a form of the one dimensional array. However, it occupies a respective sequence of storage area over all physical SLAVE processors and thus effectively forms a segment vector, that is, 2 or 3 dimensional data items. Each physical SLAVE processor may access a same specific segment without any direction attribute in spite of its position. An element of a non-directed segment array is indicated simply with a subscript index not enclosed by a pair of slashes like that of U(I).

Though no subscript indexes enclosed by a pair of slashes appear, respective segments are reserved by all physical processors.

5.3.6.1 A non-directed segment array may appear in a global or local subprogram. Every array which may appear in a local subprogram is a non-directed segment.

5.3.7 The information necessary to allocate storage for variable arrays (names, dimension bounds and direction attributes) in the source program may be provided by the DIMENSION statement.

In order to declare data types in addition to such information, an explicit type statement may be used. An explicit type statement may be used only for data type declarations.

DIMENSION statement, Explicit type statement for variable arrays
DIMENSION *a* 1 (*k* 1), *a* 2 (*k* 2), . . . , *an*(*kn*)

Type $a1$ [$(k1)$], $a2$ [$(k2)$], . . . , an[(kn)]

Where : Each a is a variable array name. Each k is an array declator. (See5. 3. 9) It is optional in an explicit type statement. *Type* is a specific type name. (See5. 2. 2)

5.3.8 *An array declarator* gives the information of dimension, dimension bounds and direction attributes. Its form is different for HOST array, directed segment array, and non-directed segment array, respectively.

5.3.8.1 The size (number of elements) of a variable array is declared by specifying, in a subscript, the number of dimensions in the array and the size of each dimension. As an exceptional case, the number of dimensions is not given explicitly for a non-directed segment array. It is implicitly defined. Each dimension is represented by an optional lower bound and a required upper bound in the form :

$e1$: $e2$ or $e2$

Where : $e1$ is the lower dimension bound. It is optional. If $e1$ (with its following colon) is not specified, its value is assumed to be 1. $e2$ is the upper dimension bound and must always be specified.

The colon represents the range of values for an array's subscript. The value of the upper bound must be greater than or equal to the value of the lower bound. The size of each dimension is equal to $e2-e1+1$. Permissible lower and upper bound ($e1$ and $e2$) are only integer constants or arithmetic expressions in which all constants and variables are of the integer type, selective permission depending on whether they appear in a HOST array declarator or in a SLAVE one, whether in a global SLAVE array declarator or a local one, or whether for a non-slashed subscript quantity or for a slashed one. A maximum number of dimensions also may be different among those array declarators.

5.3.8.2 *A HOST array declarator* is used for a HOST array. A maximum of seven dimensions is permitted.

Both upper and lower bounds may be expressions. If the array name is an actual argument, the expressions can contain only constants or constant names of integer types. If the array name is a dummy argument and is in a

subprogram, the expressions can also contain integer variables that are also dummy arguments, or appear in a common block in that subprogram.

The value of the lower bound may be positive, negative, or zero. The upper dimension bound of the last dimension of a dummy array name can be an asterisk. In this case, the dummy array is called an assumed-size array. The actual size of the last dimension is set to be equal to that of the actual array of the calling program each time the subprogram is called.

5.3.8.3 *A directed segment array declarator* is used for a SLAVE directed segment array. Either a 2- or 3-dimensional array declarator is permitted. A direction attribute is given by a specific position of a pair of slashes enclosing one or two subscript quantities in a declarator, in the same way as taken for a directed segment array element.

A slashed subscript dimension has only $e1 = 1$ as its lower bound, and hence the declarator may always omit "*e1* :". Its upper bound *e2* may be any integer constant or constant expression.

A non-slashed subscript dimension may have any integer lower and upper bound. An asterisk is permissible for a non-slashed subscript dimension, showing that each segment of the array is an assumed-size segment.

One or two asterisks also are permissible for one or two slashed subscript dimensions alternatively. It means that the segment array has the same dimension bounds as those assigned by the REGION statement in the same global subprogram.

Examples :

```
DIMENSION  U(*/,0: 33,/*)
DIMENSION  V(*,/*/)
```

5.3.8.4 *A non-directed segment array declarator* is used for a non-directed segment array.

Only one dimension is permitted. Dimension bounds are given by the same way as seen for a HOST array, or a non-slashed subscript dimension of a directed segment array declarator.

Examples :

```
DIMENSION QR(100),  SU(-1:10**2),  LF(*)
```

5.4 Record array and file array

ADETRAN allows, in addition to the usual variable arrays, record arrays and file arrays classified as SLAVE arrays. These arrays may have record elements

and file elements respectively.

5.4.1 *A record array* is a SLAVE array, each element of which is a record of the same structure. A record is an ordered set of a definite number of 'field' variables which may be of different types. Such a field variable of the record has a field name. Any numerical or logical operation is only valid for each field. For access to a field variable, it is called by the record array name with an array declarator enclosed in parenthesis, and followed by a period and field name in that order.

Example:

```
REC(I,'/J,K/). IA
```

5.4.2 *A file array* is a SLAVE array, each element of which is a file that is a sequence of records with identical structures and is of any length. Ingredient records of a file are accessed sequentially from the first to the last. If a file array, say PARTCL for example, is declared, a buffer array $PARTCL is automatically defined. It first contains all head record elements. It plays a role of a 'window' of the file array, through which record elements seen are read or new record elements are appended. For computation of record elements of a file, only records seen in the window are accessed at the current time. When the window $\$a$ is moved beyond the end of the file element a, the standard logical function EOF(a) returns the value .TRUE., otherwise .FALSE..

5.4.3 A RECORD or FILE statement is used for its declaration and gives information about record field structure, dimension, dimension size and direction attribute.

A DIMENSION statement cannot declare the dimension nor the dimension size of a record or file array. An explicit type statement cannot declare types of record field variables.

RECORD statement, FILE statement
RECORD /*a* 1 (*k* 1), *a* 2 (*k* 2), . . . [,]/[*type* 1]*x*, [*type* 2]*y*, . . . [,] /*b* 1 (*l* 1), *b* 2 (*l* 2), . . . [,]/[*type* 3]*u*, [*type* 4]*v*, . . . FILE /*a* 1 (*k* 1), *a* 2 (*k* 2), . . . [,]/[*type* 1]*x*, [*type* 2]*y*, . . . [,] /*b* 1 (*l* 1), *b* 2 (*l* 2), . . . [,]/[*type* 3]*u*, [*type* 4]*v*, . . . Where : Each *a* or *b* is a record or file array name. Each *k* or *l* is a slave array declarator of the same kind. Each group of *x*, *y*, . . . or *u*, *v*, . . . gives a set of field names of the record (of the file) *a*'s or *b*'s. Each *type* declares

the type of the following field. Without a *type*, the predefined specification or the specification by the IMPLICIT statement is applied. Each comma enclosed by [ ] is optional.

5.5 FILE-processing statements

5.5.1

RESET statement
REST (a) Where : a is a file array element.

The RESET statement resets the file window to the beginning for the purpose of reading, i. e. assigning to $\$a$ the value of the first element of a. EOF (a) becomes false, if a is not empty ; otherwise, $\$a$ is not defined, and EOF(a) remains true.

5.5.2

REWRITE statement
REWRITE (a) Where : a is a file array element.

The REWRITE statement precedes the rewriting of the file a. The current value of a is replaced with the empty file. EOF(a) becomes true and a new file element may be written.

5.5.3

GET statement
GET (a) Where : a is a file array element.

The GET statement advances the file window $\$a$ to the next component ; i. e. assigns the value of this component to the buffer variable $\$a$. If no next component exists, then EOF(a) becomes true and the resulting value of $\$a$ is not

defined. The effect of GET(a) is defined only if EOF(a) is false prior to its execution.

5.5.4

PUT statement
PUT (a) Where : a is a file array element.

The PUT statement appends the value of the buffer $\$a$ to the file array element a. The effect is defined only if prior to execution the predicate EOF(a) is true. EOF(a) remains true and $\$a$ becomes undefined.

6. Common data area

In order to cause sharing of storage by two or more program units, a COMMON statement is used. It may specify the names of variables and variable arrays that are to occupy this area. In order to allow sharing of storage by two or more variable array elements in a single program unit, an EQUIVALENCE statement is used.

6.1 Common block

A common data area shared by two or more program units is called a *common block*. A common block is valid either among HOST subprograms, or among global subprograms, or among local subprograms. As an exceptional case, a HOST array may appear in a named common block shared by a HOST subprogram and a GSUBROUTINE subprogram.

Names of variables or variable arrays in a common block may be different among those programs sharing the common block.

There are two kinds of common block : blank common and named common.

6.1.1 *A blank common block,* at most one, is permissible in a program unit. In a SLAVE subprogram, variables or non-directed segment arrays may appear in a blank common block.

HOST variables or variable arrays declared in a blank common block cannot be initialized by a BLOCKDATA subprogram.

6.1.2 Several *named common blocks* may appear in a program unit. Each block has a specific name and occupies a distinct storage area. Each program unit which uses the given named common block must define it to be of the same

length.

A common block containing directed segment arrays shared among global subprograms must be named. Moreover, those with the same dimension, the same direction attribute, or those without any direction attribute must be listed under a named common block. HOST variables and variable array elements in a named common may be assigned initial values by DATA statements in a BLOCKDATA, while any SLAVE variables and arrays cannot be initialized by DATA statements.

6.1.3 A COMMON statement declares common blocks.

If a variable array is followed by an array declarator in a common block, its dimension or dimension size need not be specified by another DIMENSION statement.

COMMON statement
COMMON /*r*1/*a*11[(*k*11)], *a*12[(*k*12)], . . . [,]/*r*2/ /*a*21[(*k*21)], *a*22[(*k*22)], . . . [,]/*rn*/*an*1 [(*kn*)], *an*2 [(*kn*2)], . . . Where: Each *a* is a variable name, variable (neither RECORD nor FILE) array that is not a dummy argumment. Each *k* is an array declarator. It is optional only in a HOST subprogram. It must exist in a SLAVE subprogram, and gives dimension and direction attribute information. Its form is the same as in the DIMENSION statement (See 5. 3. 9). Each *r* represents an optional common block name. These names must always be enclosed in slashes.

The form // (with no characters except possibly blanks between slashes) may be used to denote blank common. If *r1* denotes the blank common, the first two slashes are optional.

Variables or variable array names listed for a named or blank common block name must have a common attribute, as mentioned in 6.1.2. A comma preceding a common block name /*r*/ may be omitted.

Examples:

```
COMMON  A(10)/X/U(256,/256/)
COMMON  /Y/G(/32/,32)
```

6.2 Equivalenced storage area

6.2.1 A storage area shared by two or more variables or variable array elements is called an *equivalenced storage area.* An equivalenced storage area may be shared by all these variables or variable array elements which are in anyone group of the followings:

a) HOST variables, HOST arrays (non character)
b) HOST character strings
c) SLAVE variables, non-directed segment array elements
d) SLAVE directed segment array elements with the same dimension and the same direction attribute.

Any two elements in a common block, or in different common blocks can not be placed in equivalence.

6.2.2 An EQUIVALENCE statement specifies equivalence storage areas.

EQUIVALENCE statement
EQUIVALENCE (*a*11, *a*12, *a*13, . .), (*a*21, *a*22, *a*23, . . .), . .
Where: Each *a* is a variable, variable array or variable array element name.

All the elements with in a single pair of parenthesis share the same storage locations in a single program unit. An array name without subscript in an EQUIVALENCE statement is interpreted to refer to its first element. Since arrays are stored in a predetermined order, equivalencing two elements of two different arrays may implicitly imply equivalencing other elements of the two arrays. The EQUIVALENCE statement must not contradict itself or any previously established equivalences.

A variable in a program unit can be made equivalent to a variable in a common block. If the variable that is equivalent to the variable in the common block is an element of an array, the implicit equivalencing of the rest of the elements of the array can extend the size of the common block. The size of the common block cannot be extended so that elements are added before the beginning of the established common block.

Examples:

```
DIMENSION     X(16/,32,/16),  Y(16/,0:33,/16)
EQUIVALENCE  (X(/,,/),  Y(/,1,/))
```

7. SLAVE system specification

In a global subprogram, it is necessary to specify configuration of the

SLAVE system prior to other specifications, except PARAMETER statements.

7.1 Parallel index region

A range covered by the logical processor array is called a *parallel index region*.

7.1.1 A REGION statement specifies an actual configuration of logical processors and a range of processor numbers.

REGION statement	
REGION (*r*1, *r*2)	(1)
REGION (*r*1, *r*2, *r*3)	(2)
REGION (*r*1,)	(3)
REGION (, *r*2)	(4)
Where: Each *r* is an integer constant or constant expression whose value is greater than or equal to 1. It gives an upper bound of its corresponding processor number index. (Each lower bound is assumed to be 1.)	

7.1.2 The forms (1) and (2) specify standard configurations of logical prosessors for processing segment arrays.

The form (1) specifies two configurations of 1-dimensional processor arrays and that the first processor number which may be enclosed in slashes has an upper bound *r1* and the second processor number which also may be enclosed in slashes has an upper bound *r2*.

If a global subprogram is required to process a segment array, say U(/I/, J), I range being from 1 to *r1*, and another segment array, say V(I, /J/), J range being from 1 to *r2*, the global subprogram must contain the REGION statement (1).

The form (2) specifies three configurations of 2-dimensional processor arrays and that the first slashed processor number has an upper bound *r1*, the second *r2* and the third *r3*.

7.1.3 The forms (3) and (4) specifies a 1-dimensional configuration of logical processors for processing 2-dimensional segment arrays with a fixed direction attribute. Such a configuration is applied for matrix computations. In processing a segment array, say U(/I/, J) instead of a $r1 \times r1$ matrix, the specification REGION (*r1*,) is necessary.

8. Expressions

An expression is a rule for calculating a value where the conventional rules of left to right evaluation and operator precedence are observed.

8.1 Kinds of expressions

ADETRAN provided four kinds of expressions: arithmetic, relational, logical and character. Character expressions are only valid in a HOST FORTRAN program unit. We here omit its details.

8.1.1 *An arithmetic expression* has some operands of numerical data, which are combined with the following arithmetic operators:

Precedence	Definition	Operator
1	Exponentiation	**
2	Multiplication	*
	Division	/
3	Addition	+
	Subtraction	−

8.1.2 *A relational expression* is formed by combining two arithmetic expressions with a relational operator, or two character expressions with a relational operator. It expresses a condition that can be either true or false.

There are six relational operators, each of which must be preceded and followed by a period, and which are as follows:

Precedence	Definition	Operator
4	$>$	.GT.
	$\geqq$	.GE.
	$<$	.LT.
	$\leqq$	.LE.
	$=$	.EQ.
	$\neq$	.NE.

8.1.3 *A logical expression* has some operands of logical data, which are combined with the logical operators, each of which must be preceded and followed by a period, and which are as follows:

Precedence	Definition	Operator
5	Negation	.NOT.
6	Logical product	.AND.
7	Logical sum	.OR.
8	Equivalence	.EQV.
	NOT equivalence	.NEQV.

8.2 HOST and SLAVE expressions

No expressions containing both HOST and SLAVE data together are permissible.

8.2.1 *A HOST expression* has only HOST data as operands in a HOST program unit. Each operand must be a scalar. No array itself appears in a HOST expression.

8.2.2 *A SLAVE expression* has only SLAVE data as operands in a SLAVE program unit. When a constant, variable, or non-directed segment array element appears as an operand, it seems to be a scalar, but it is really a vector whose components are shared by all physical processors.

In processing a record or file array, only field variables of each record of the array element may appear in an expression. A record or file array element itself cannot appear in an expression, and much less such an array itself.

9. Assignment statement

An assignment statement serves to replace the current value of a variable by a new value indicated by an expression.

Assignment statement
a=*expression* Where: a is a variable, variable array element or field variable of a record/file array element. *expression* is an arithmetic or logical expression.

A HOST assignment statement may appear in a HOST program unit, and may contain only HOST data.

A SLAVE assignment statement may appear in a SLAVE program unit, and may contain only SLAVE data. In a global subprogram, all directed segment

array elements on both sides of an assignment operator(=) are of the same dimension, direction attribute and processor number indexes. In a global subprogram, any variable uni-vector cannot appear on the left side of an assignment operator in any parallel paragraph.

Example:

U(I, /J, K/)=A(I+1)+B*$FC(I, /J, K/). IA

10. General control statements

We call all FORTRAN statements which control flow of computation *general control statements*. They all are also valid in ADETRAN.

10.1 GO TO statement

GO TO statements permits transfer of control to an executable statement specified by the number in the GO TO statement. The GO TO statements are:

1) Unconditional GO TO statement
2) Computed GO TO statement
3) Assigned GO TO statement

Computed GO TO and assigned GO TO statements are permitted only in a HOST program. Their details are omitted.

An unconditional GO TO statement causes control to be transferred to the statement specified by the statement number without any condition.

Unconditional GO TO statement
GO TO *x* Where: *x* is the number of an executable statement in the same program unit.

10.2 IF statements

IF statements control computation flow depending on some conditions. Related statements are:

1) Arithmetic IF statement
2) Logical IF statement
3) Block IF statement
4) END IF statement
5) ELSE IF statement
6) ELSE statement

These statements may appear in a HOST program, in a global subprogram or in a local subprogram.

10.2.1

Arithmetic IF statement
IF (m) $x1$, $x2$, $x3$ Where: m is an arithmetic expression of any type except complex.Each x is the number of an executable statement in the program unit containing the IF statement.

The arithmetic IF statement causes control to be transferred to the statement numbered $x1$, $x2$, or $x3$ when the value of the arithmetic expression m is less than, equal to, or greater than zero, respectively. Data which may appear in the expression m and statement number xi are as follows:

1) In HOST program, m may contain any constant, variables or array elements of an integer or real type. xi may be a statement number of any executable statement in the same HOST program unit.
2) In a global subprogram and moreover outside of any parallel paragraph, m may contain any constants or variable uni-vectors. xi may be the statement number of an executable statement not contained in any parallel paragraph or DO paragraph (which does not contain the arithmetic IF statement).
3) In a global subprogram and moreover in a parallel paragraph, m may contain any constants, variables, and non-directed or directed variable segment array elements of an integer or real type. xi may be the statement number of an executable statement which is in the same parallel paragraph, and moreover in the DO paragraph that contains the arithmetic IF statement or outside of any DO paragraphs.
4) In a local subprogram, m may contain any constants variables, or non-directed segment elements of an integer or real type. xi may be the statement number of an executable statement in the same program unit.

10.2.2

Logical IF statement
IF (m) st Where: m is any logical expression. st is any executable statement except a DO statement, another logical IF statement, an END statement, a block IF statement, an ELSEIF statement, an ELSE statement, or any parallel control statement.

The logical IF statement is used to evaluate the logical expression *m* and execute or skip statement *st* depending on whether the value of the expression is true or false, respectively.

Data which may appear in *m* are as follows:

1) In a HOST program, *m* may contain any constants, variables, or array elements of the logical type.
2) In a global subprogram and moreover outside of any parallel paragraph, *m* may contain any constants or variable uni-vectors of a logical type.
3) In a global subprogram and moreover in a PDO paragraph, *m* may contain any constants, variables, or non-directed or directed variable segment array elements of the logical type.
4) In a local subprogram, *m* may contain any constants, variables, or non-directed segment elements of the logical type.

10.2.3

Block IF statement, END IF statement

IF (*m*) THEN
......
END IF
 	
Where : *m* is any logical expression.

The block IF statement is used with the ENDIF statement and, optionally, the ELSE IF and ELSE statements to control the execution sequence. An IF-block begins with the first statement after the block IF statement (or an ELSE IF statement) ends with the statement preceding the corresponding ENDIF statement, and includes all the statements in between. It may contain another IF-block which follows an ELSE IF statement.

Execution of a block IF statement evaluates the expression *m*. If the value *m* is true, normal execution sequence continues with the first statement of the IF-block, which is immediately following the block IF statement. If the value of *m* is true, and the paragraph to follow the block IF statement immediately is empty, control is transferred to the corresponding ENDIF statement. If the value of *m* is false, control is transfered to the next ELSEIF, ELSE, or ENDIF statement.

Rules for constructing an IF-block and data which may be in *m* are as follows:

1) In a HOST subprogram, an IF-block may be constructed arbitrarily. *m* may contain any constants, variables or array elements of the logical type.

2) In a global subprogram, any IF-block either contains some parallel paragraphs completely or must be in a parallel paragraph. In the former case m may contain any constants or variable uni-vectors, and in the latter case m may contain any constants, variables, or non-directed variable segment array elements of the logical type.
3) In a local subprogram, an IF-block may be constructed arbitrarily. m may contain any constants, variables or non-directed segment elements of the logical type.

10.2.4

ELSEIF statement
ELSEIF (m) THEN Where: m is any logical expression.

The ELSEIF statement is executed if the preceding block IF condition is evaluated as false, and constructs another inner IF-block.

If the value of the logical expression m is true, normal execution sequence continues with the first statement of the inner IF-block.

If the value of m is true and the paragraph to follow the ELSEIF statement is empty, control is transferred to the corresponding ENDIF statement.

If the value of m is false, control is transferred to the next ELSEIF, ELSE, or the corresponding ENDIF statement.

Data which may appear in m are the same as those mentioned in 10.2.3.

10.2.5

ELSE statement
ELSE

The ELSE statement is executed if the preceding block IF or ELSEIF condition is evaluated as false. Normal execution continues.

10.3 Repetitive statement

10.3.1 A DO statement is used for repetition of a sequence of statements.

DO statement
DO x[,]i=$m1$, $m2$ [,$m3$] Where: x is the number of an executable statement in the same program unit as

the DO statement, that denotes the end of the DO paragraph. The comma after x may be omitted. i is an integer variable (not an array element) called the DO variable. $m1$, $m2$, and $m3$ are integer expressions that define the DO-paragraph iteration: $m1$ is an initial value, $m2$ is a test value and $m3$ is an increment. $m3$ is optional and cannot be zero. If it is omitted, its value is assumed to be 1. In this case the preceding comma must also be omitted.

The DO statement is a command to execute, at least once, the statements that physically follow the DO statement, up to and including the statement numbered x. These statements constitute the *DO paragraph.*

The statements in the DO paragraph are executed only when either $m1 \leqq m2$, $m3 > 0$ or $m1 \geqq m2$, $m3 < 0$.

If one of the above relationships is true, the first time the statements in the DO paragraph are executed, i is initialized to the value $m1$, and on each succeeding iteration, i is increased by the value $m3$.

The number of iterations that can be executed, called the iteration count, is the value of $\mathrm{MAX}((m2-m1+1)/m3,\ 0)$. When the iteration count is zero, execution continues with the statement following the last statement of the DO paragraph, or the next outer DO if the statement number x is shared by more than one DO statements.

10.3.2 In a HOST program unit, there may be other DO paragraphes within the DO paragraph. All statements in the inner DO paragraph must be in each outer DO paragraph. A set of DO statements satisfying this rule is called a nest of DO's.

In a SLAVE local subprogram, no DO nest is permissible.

10.3.3 In a global subprogram, either a DO paragraph contains several parallel paragraphs, or a DO paragraph is fully contained in a parallel paragraph. In the former case, the integer expressions $m1$, $m2$, and $m3$ may contain only integer constants or variable uni-vectors. In the latter case, $m1$, $m2$, and $m3$ may contain integer constants, variables or non directed segment elements.

10.4 General punctuation statements

10.4.1 A CONTINUE statement is a statement that may be placed anywhere in the source program (where an executable statement may appear) without affecting the seguence of execution. Its format is the same as in FORTRAN.

It may be used as the last statement in a DO paragraph in order to avoid ending the DO paragraph with an unconditional or assigned GO TO, block IF, ELSEIF, ELSE, ENDIF, STOP, RETURN, END, arthmetic IF, another DO statement, or a logical IF statement containing an unconditional or assigned GO TO, or STOP, RETRUN, an arithmetic IF statement.

10.4.2 A PAUSE statement temporarily halts execution of a HOST object program. It cannot be used in a SLAVE program.

10.4.3 A STOP statement terminates execution of a HOST object program. It cannot be used in a SLAVE program.

10.4.4 An END statement terminates a program unit. Physically, it must be the last statement of each program unit. Its format is the same as in FORTRAN.

11. Parallel control statements

Parallel control statements are those of ADETRAN PROPER.

These statements are placed in a global subprogram for control of the whole ADENA SLAVE system.

11.1 PDO Statement

PDO statement, PEND statement
PDO *sr*[, * (*asub*)] … (PDO paragraph) PEND Where : *sr* is a parallel processing subscript (or processor number) range. (See 11. 1. 1) * (*asub*) is optional, and is used to avoid repetition of the same parallel processing subscripts for directed SLAVE arrays in the subsequent statements in the PDO paragraph. *asub* is composed of the same subscript indexes as those in *sr*, enclosed in slashes and an empty separated by commas. It is called repeat avoidance option.

The word 'PDO' is a short from of 'PARALLEL DO'.

The PDO statement commands those logical processors whose numbers are in the processor number region *sr* to execute, in parallel, those statements that physically follow the PDO statement, up to the following PEND statement. We call such set of statements a *PDO paragraph*. It is a typical parallel paragraph.

11.1.1

Parallel processing subscript (processor number) range	
$i=m1,\ m2$	(1)
$i=m1,\ m2,\ j=n1,\ n2$	(2)
(Short forms)	
$i,\ j=m1,\ m2$	(3)
Where: Each i, or j is a parallel processing subscript (processor number) index. Each $m1$, or $n1$ is a respective lower bound, and each $m2$, or $n2$ is a respective upper lound. Such bounds must be integer constant expressions whose values are greater than 1.	

The short form (3) is permissible when $n1=m1$ and $n2=m2$ hold in (2).

11.1.2 Followings are some comments in using a PDO statement.

1) In processing some directed segment arrays in a PDO paragraph, they all have the same dimension, the same direction attribute, and the same processor number indexes or list vector element indexes enclosed in parentheses in slashes. There may, in addition, appear some variables or non-directed segments.

Example:

```
      PDO  J= 1,32
      DO  10  I= 1,64
   10 B(I,/J/)=A(I,/J/)+C(I)
      PEND
```

2) For the PDO paragraph in which only directed segment arrays and variables appear but no non-directed segment arrays, slashed indexes can be omitted in segment array elements by putting the omitted common indexes in the form * (asu*b*) at the rear part of the PDO statement.

Example The paragraph

```
      PDO  J,K= 1, N
      DO  10  I= 1, N
   10 V(I,/J,K/)=U(I+1,/J,K/)+U(I-1,/J,K/)
      PEND
```

can be equivalently written as

```
    PDO  J,K=1,N, * (,/J,K/)
    DO  10  I=1,N
10  V(I)=U(I+1)+U(I-1)
    PEND
```

3) A PDO paragraph cannot contain another parallel paragraph.
4) No transfers of control into and out of a PDO paragraph are permissible.
5) Call of, and retrun from, a LSUBROUTINE from within a PDO paragraph are permissible.
6) The indexing parameters of the PDO statement, *i, m1, m2,* etc., must not be changed by any statement within the PDO paragraphe.

11.2 Parallel accumulation operation

In some computations, such as a mathematical inner product computation, it happens to be better to accumulate several partial results computed from those logical segments associated with a physical processor to get each partial sum as a physical array element without any data transfer among different physical processors.

In such a case, it is necessary to associate each logical subscript enclosed in slashes with a corresponding physical one. This association is given by doubled slashes, as seen in //I// or //J,K//, where //I// or //J,K// means a physical subscript corresponding to a logical subscript /I/ or /J,K/, respectively.

Example:

```
    PDO  J=1,256
    V(1, //J//)=0.0
    PEND
    PDO  J=1,512
    DO  10  I=1,64
10  V(1, //J//)=V(1, //J//)+W(I,/J/)
    PEND
```

The result value of V(1, //J//) would be the sum

$$\sum_{I=1}^{64} W(I,/J/)+W(I,/J+256/).$$

11.3 PASS statement

A PASS statement is used for editing a segment array with a direction attribute into that with another direction attribute. It also may be used for

transferring a host array into a segment array or vice versa. This is just data transfer between the HOST computer and the SLAVE system.

11.3.1 Contents which a PASS statement commands are some replacement statements. A replacement is an assignment statement of a specific form, i. e. of the form that both hands of an equal sign have array names of the same dimension and same indexes but a different attribute, HOST vs. SLAVE, or a direction vs. another direction.

Examples:

```
U(/I/, J)=U(I, /J/)
U(I, J, K)=V(I/, J, /K)
U(/I, J/)=U(I, J)
```

A set of replacement statements which are of the same kind (HOST or SLAVE), and the same direction attributes, if any, on the respective hands of equal signs are called *a PASS paragraph*. It also is a typical parallel paragraph.

Example

```
┌U(I, /J/)=U(/I/, J)
│V(I, /J/)=V(/I/, J)
└W(I, /J/)=X(/I/, J)
```

11.3.2 A PASS and PEND statement punctuate a PASS paragraph.

PASS statement, PEND statement
PASS *pr*
...... (PASS paragraph)
PEND
Where: *pr* is a PASS processing subscript range.

Permissible forms of the PASS processing subscript range are as follows:

PASS processing subscript range	
$i=m1, m2, j=n1, n2$	(1)
$i=m1, m2, j=n1, n2, k=l1, l2$	(2)
(short forms)	
$i, j=m1, m2$	(3)

$i=m1, m2, j, k=n1, n2$ etc (4)

$i, j, k=m1, m2$ (5)

Where: i, j, or k is a PASS processing subscript variable, and it is of an integer type. $m1, n1$ or $l1$ is a lower bound of a corresponding subscript, and $m2$, $n2$, or $l2$ is an upper bound. These are integer constant expressions with positive values.

The short form (3) is permissible when $n1=m1=$ and $n2=m2$ hold in (1). The short form (4) given is permissible when $l1=n1$ and $l2=n2$ hold in (2). The short form (5) is permissible when $l1=n1=m1$ and $l2=n2=m2$ hold in (2). The PASS processing subscript range *pr* may be omitted when the range happens to be the same as that defined by the REGION statement in the same program unit. With omission, all subscripts may be replaced by asterisks when their corresponding ranges would match with the REGION range.

Example:

```
PASS
U(*,/*,*/)=U(/*,*/,*)
PEND
```

11.4 PHASE statement

Suppose that, in processing segment arrays with a fixed direction attribute, the logical SLAVE processors with a predefined linear (or bi-directional linear) order hope to get the segment with a specific name from each preceding processor, execute some computation and then send the segment with the same name to each following processor, one upon another. Such processing may, for example, appear for solving a linear equation system with an upper or lower triangular matrix coefficient. The matrix is considered as a two-dimensional segment array with a fixed direction attribute (either row or column ordering).

A PHASE statement allows sending segment components computed immediately to the next processor, one upon another, before the complete computation of each segment.

PHASE statement, PEND statement

PHASE (*ps*) *sr*

...... (PHASE paragraph)

PEND

Where : *ps* is a transferred segment. *sr* is a parallel processing subscript (processor number) range.

11.4.1 The transferred segment is composed of one or two segment element names and an element range, separeted by a comma.

Transferred segment

$a(i), i=l1, l2$ [, −1]

$a(i), b(i), i=l1, l2$ [, −1]

Where : a or b is a non-directed segment name. i is a subscript variable of an integer type. $l1$ and $l2$ are an initial and final value of the subscript variable, respectively.

They are integer constant expressions. '−1' with a preceding comma is optional, and it must be given for the case that the segment elements are determined in the decreasing order.

11.4.2 The parallel processing subscript range in the PHASE statement may take forms similar to those in a PASS statement, but there are some differences between them. First, when two subscripts appear, their listed order has the definite meaning that the first subscript may change faster than the second. This also defines the kind of transferring segment elements, that is, the first transfered segment to the first subscript direction, and the second to the second direction. Secondly, the subscript variable may change either in the increasing order or in the decreasing order.

Parallel processing subscript (processor number) range in the PHASE statement

$j=m1, m2$ [, −1] (1)

$j=m1, m2$ [, −1], $k=n1, n2$ [, −1] (2)

(Short form)

$j, k=m1, m2$ [, −1] (3)

Where : j or k is a parallel processing subscript (processor number) variable.

$m1$ or $n1$ is an initial value of the respective subscript, and $m2$ or $n2$

is a final value. For the case of $m1 > m2$ or $n1 > n2$, '−1' must be placed with a preceding comma.

The short form (3) can be used when $n1 = m1$ and $n2 = m2$ hold in (2). Also in this case, the order of index i and j is essential.

11.4.3 A typical PHASE paragraph is as follows:

```
   DO 10 I=1, N
10 V(I)= ....        (Initial values settled)
   PHASE (V(I), I=1,N) J=1, M
   ......           (A)
   DO 20 I=1, N
   F(I,/J/)=....+V(I)+..(Use of the preceding segment)
20 V(I)=F(I,/J/)
   ......           (B)
   PEND
```

In part (A) or (B), any statements permissible in a PDO paragraph may be placed, but no replacement of V(I) may happen in part (B).

In the execution of PHASE statement with the settled initial values of V(I) only the data corresponding to J= 1 become usable as if they were to be sent from the (J=) 0-th processor, and all other segment V's become unable. Their segments become usable just after getting from each preceding processor and return to be unable after sending each result value to each following processor.

11.5 POUR Statements

It is POUR statement to command to copy a specific vector whose components are of elements with the same element number of a two-dimensional directed segment array to a segment with a specific segment number or to segments with those numbers in a range. Copied segments are of the same direction attribute as that of the original segment array from which the vector was extracted.

POUR statement, PEND statement
POUR *pr* ...... (POUR paragraph) PEND

Where : *pr* is a POUR processing subscript range.

11.5.1 The POUR paragraph is composed of several replacement statements having segment array elements with the same dimension and direction attribute on both sides of the equal signs. The index variable outside slashes have a definite value on the right hand side of each equal sign.

The processor number variable (or index variable of a list vector) in slashes on the right hand side must appear as the segment element index variable outside of slashes on the left hand side. The processor number index in slashes on the left hand side may be a specifc value or a variable whose range is specified by the POUR statement.

Examples :

```
V(/1/, J)=W(/J/, 1)
V(/I/, J)=W(/J/, 2)
```

11.5.2

POUR processing subscript range	
$i=m1, m2$	(1)
$i=m1, m2, j=n1, n2$	(2)
(Short form)	
$i, j=m1, m2$	(3)
Where : Each i or j is a subscript variable of an integer type. $m1$ or $n1$ is a lower bound of each subscript and $m2$ or $n2$ is each upper bound.	

The form (1) is applicable for the case that the common subscript variable has a range, and other subscripts are of fixed values, i. e. that the new segment is a specific one.

The form (2) is applicable for the case that a vector is copied to several segments.

The short form (3) may be used when $n1=m1$ and $n2=m2$ hold in (2).

11.5.3 A POUR statement may be used for usual matrix computations. Suppose that all row vectors of a matrix A and all components of a column vector u are shared by processors, and it is required to find their product Au. For parallel processing in the way that each product of a row vector u is computed by each processor, the column vector u must be fist transferred to all

processors having respective rows of A. This is realized by the POUR statement.
Example:

```
     POUR  I,J=1,N
     UT(/I/,J)=U(/J/,1)       (transpose column to row)
     PEND
     PDO
     C(/I/,1)=0.0
     DO  10  J=1,N
  10 C(/I/,1)=C(/I/,1)+A(/I/,J)*UT(/I/,J)  (scalar product)
     PEND
```

11.6 PAD statement

A PAD statement is used to copy a 2-dimensional directed segment with a specific number to several segments with the same direction attribute.

PAD statement, PEND statement
PAD *pr* . . . . . (PAD paragraph) PEND Where: *pr* is a PAD processing subscript range.

11.6.1 The PAD paragraph is composed of several replacement statements with elements of two dimensional segment arrays with the same direction attribute on both hand sides of the equal signs.

Each processor number subscript (or list vector element subscript) in slashes on the right hand side takes a fixed value. Those subscript variables outside of slashes on both sides are common.

Example:

```
B(/I/,J)=A(/1/,J)
```

11.6.2

PAD processing subscript range	
$i=m1, m2$	(1)
$i=m1, m2, j=n1, n2$	(2)

(Short form)

$i, j = m1, m2$ (3)

Where : Each i or j is a PAD processing subscript variable, $m1$ and $n1$ are respective lower bounds, and $m2$ and $n2$ are respective upper bounds. These bounds may be integer constant expressions.

The form (1) is permissible for the case in which the addressed segment is only one, and the variable i indicates the segment elements. The short form (3) is permissible when $n1 = m1$ and $n2 = m2$ hold in (2).

11.6.3 A PAD statement may be used for such data transfer as seen in the Gauss elimination method, i. e. sending the pivot row to others.

```
PAD  I, J= 1 , N
B(/I/, J)=A(/1/, J)
PEND
```

11.7 PUSH statement

A PUSH statement is supplied for 'particle push' processing in particle simulation codes.

The PUSH statement commands to edit a file array whose element records have some address fields so that all records may become new ingredients of a respective by addressed file array element. Each record, of couse, corresponds to each 'particle' of simulation.

PUSH statlmeut
PUSH ($a=b$, pr) Where : a is the new file array element whose subscripts are corresponding address field variables of the original file array. b is the original file array element assigned by the PUSH processing subscript variables. pr is a PUSH processing subscript range.

a and b must have the same dimensions and direction attributes. The PUSH processing subscript range pr is of the same form as that of the PASS processing subscript range.

Example :

PUSH(C(IA, /JA, KA/) = B(I, /J, K/), I, J, K = 1, N)

Where IA, JA, and KA are the address field variables of the ingredient record of the file array B.

12. SLAVE system synchronizing statements

For synchronizing the whole SLAVE system, IFALL and IFANY statements are supplied. The IFALL statement is to check first whether all conditions, each of which is seen by each processor, are satisfied or not and then to take a branch control. The IFANY statement is to check first whether anyone of the conditions seen is satisfied or not and then to take a branch control. IFALL/IFANY statement may only appear in a global subprogram.
There are some related statements.
1) Logical IFALL/IFANY statement
2) Block IFALL/IFANY statement
3) ELSEA statement
4) ELSEIFALL/ELSEIFANY statement
5) ENDIFA statement

12.1 Logical IFALL/IFANY Statement

Logical IFALL/IFANY statement
IFALL (*dlist*) *st* IFANY (*dlist*) *st* Where : *dlist* is an implied logical data list. *st* is either unconditioal GOTO, or RETURN statement.

The logical IFALL statement commands to execute the statement *st* or to skip it depending on whether the result value of logical AND of the listed data is true or false, respectively.

The logical IFANY statement commands to execute the statement *st* or to skip it depending on whether the result value of logical OR of the listed data is true or false, respectively.

12.1.1

Implied logical data list	
[.NOT.] *name* $(i,/j/)$, $i=m1, m2$, $j=n1, n2$	(1)
[.NOT.] *name* $(/i/, j)$, $i=m1, m2$, $j=n1, n2$	(2)
[.NOT.] *name* $(i,/j, k/)$, $i=m1, m2$, $j=n1, n2$, $k=l1, l2$	(3)
[.NOT.] *name* $(i/, j,/k)$, $i=m1, m2$, $j=n1, n2$, $k=l1, l2$	(4)
[.NOT.] *name* $(/i, j/, k)$, $i=m1, m2$, $j=n1, n2$, $k=l1, l2$	(5)
(Short form)	
[.NOT.] *name* $(i,/j/)$, $i, j=m1, m2$ etc.	(6)
[.NOT.] *name* $(i,/j, k/)$, $i=m1$, $m2$, $j, k=n1, n2$ etc.	(7)
[.NOT.] *name* $(i,/j, k/)$, $i, j, k=m1, m2$ etc.	(8)
(Element specified form)	
[.NOT.] *name* $(c,/j/)$, $j=n1$, $n2$ etc.	
[.NOT.] *name* $(c,/j, k/)$, $j=n1$, $n2$, $k=l1$, $l2$ etc.	
Where : *name* ( ) is an array element. i, j, or k is a subscript variable of the integer type or an asterisk. $m1, n1$ and $l1$ are lower bounds, and $m2$, $n2$ and $l2$ are upper bounds of the corresponding subscripts. Those bounds are integer constant expressions. c is an integer constant.	

Short forms are applied when the respective lower and upper bounds happen to match each other. For example, the short form (6) is permissible when $n1=m1$ and $n2=m2$ hold in (1).

Element specified forms are used when the non-slashed variable is replaced with an integer constant.

Each .NOT. is optional. With it, each array element is negated. If asterisks are placed instead of processor number subscripts in slashes, the corresponding ranges are considered to be the same as those given in the REGION statement, and those range expressions may be omitted.

12.1.2 An IFALL/IFANY statement is applied for a convergence check of an iterative mothod.

Example :

```
IFALL(OK(I,/*,*/), I=N) GOTO 20
```

or equivalently

```
IFANY(.NOT.OK(I, /*, */), I=1, N) GOTO 20
```

12.2 Block IFALL/IFANY statement and ENDIFA statement

Block IFALL/IFANY statement also command a branch control depending upon whether .AND. /.OR. of listed data is true or false.

If the result is true, the following statements up to the next ELSEA, or ELSEIFALL/ELSEIFANY statement are executed. If the result is false, those statements following the ELSEA statement and up to the corresponding ENDIFA statement are executed or a further branch control is taken by the ELSEIFALL /ELSEIFANY statement. If such ELSEA or ELSEIFALL/ELSEIFANY does not appear, those statements following the first block IFALL/IFANY statement and up to the corresponding ENDIFA statement are skipped.

Not depending on whether an ELSEA or ELSEIFALL/ELSEIFANY statement may appear or not, the corresponding ENDIFA statement must be placed to close the IFALL/IFANY block.

Block IFALL/IFANY statement, ENDIFA statement
IFALL (*dlist*) THEN . . . . . . (IFALL block) IFANY (*dlist*) THEN . . . . . . (IFANY block) ENDIFA Where : *dlist* is an implied logical data list. (See 12. 1. 1)

We call the set of statements after the IFALL/IFANY statement (or alternatively an ELSEIFALL/ELSEIFANY statement mentioned below in 12.3) up to the corresponding ENDIFA statement *an IFALL/IFANY block.* An IFALL /IFANY block may contain another IFALL/IFANY block which follows an ELSEIFALL/ELSEIFANY statement.

12.3 ELSEIFALL/ELSEIFANY statement

An ELSEIFALL/ELSEIFANY statement makes an inner IF block when the preceding condition is false.

ELSEIFALL/ELSEIFANY statement
ELSEIFALL (*dlist*) THEN ELSEIFANY (*dlist*) THEN Where : *dlist* is an implied logical data list. (See 12. 1. 1)

In execution of the IFALL (or alternatively IFANY) statement, logical .AND. (or .OR.) of listed data is first computed. Depending on whether the result value is true or false, the following branch control is taken. If it is true, then those statements following the ELSEIFALL (or ELSEIFANY) statement and up to the next ELSEA or ELSEIFALL/ELSEIFANY statement are executed. Without such ELSEA or ELSEIFALL/ELSEIFANY statement, those up to the corresponding ENDIFA statement are executed.

If the result value is false, those statements following the next ELSEA statement are executed or another branch control is taken by the next ELSEIFALL/ELSEIFANY statement. Without such an ELSEA or ELSEIFALL/ ELSEIFANY statement, the IFALL/IFANY block is skipped.

12.4 ELSEA statement in IFALL/IFANY block

This ELSEA statement has a role similar to the usual ELSE statement in an IF block.

ELSEA statement
ELSEA

It indicates the starting point of the paragraph to be executed when the condition in the preceding IFALL/IFANY or ELSEIFALL/ELSEIFANY statement is false. These statements following the ELSEA statement and up to the next ENDIFA statement are then executed.

13. FUNCTION

To use a function, it is necessary to define the function previously (i. e., specify which calculations are to be performed), and to refer to the function by name where required in the program. The definition of a function is expressed by its name, dummy arguments and procedure for evaluating the function. There are three kinds of function definitions: statement function, FUNCTION subprogram and LFUNCTION subprogram.

When the name of a function, followed by a list of its arguments, appears in

any expression, it refers to the function and causes the computations to be performed as indicated by the function definition. The resulting quantity (the function value) replaces the function reference in the expression and assumes the type of the function.

13.1 Statemant function statement

A statement function definition specifies operations to be performed whenever that statement function name appears as a function reference in another statement in the same program unit.

Statement function statement
name (*a* 1 , *a* 2 , . . . , *an*)=*expression*
Where : *name* is the statement function name. Each *a* is a statement function dummy argument. It must be a distinct variable. There must be at least one dummy argument. *Expression* is any arithmetic or logical expression.

All statement function definitions to be used in a program must follow the specification statements and precede the first executable statement of the program.

13.1.1 In a HOST program, a character expression also is permissible as *expression.*

The expression of a statement function cannot refer to the statement function name itself. When the expression refers to another statement function, that function must be defined previously. A statement function definition may have such arguments which appear as dummy arguments of the subprogram containing the statement function, but their values must be defined prior to its reference.

In a statement function of a HOST program, each dummy argument is considered to be a HOST variable scalar. In a statement function of a SLAVE subprogram, each dummy argument is considered to be a SLAVE variable vector.

13.1.2 An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument. The actual argument of a statement function reference must not be changed by the evaluation of the expression of that statement function. That is, an argument of a function that is changed by its evaluation cannnot also be an argument of a

statement that references that function. If there appear some SLAVE arrays as actual arguments, their dimensions, direction attributes and subscript variables in slashes are the same, respectively.

13.1.3 *Examples*:

```
(Definition)           (Reference)
WA(P,Q)=P+Q            AIM=CAT-WA(S1,S2)
GFU(U,V)=1.0-U*V       BEM=A+GFU(W(I+1,/J/), W(I,/J/))
```

13.2 FUNCTION and LFUNCTION Subprogram

A FUNCTION subprogram is a kind of HOST subprogram consisting of a FUNCTION statement followed by other statements including at least one RETURN statement. It is executed whenever its name is referred to in another program. An LFUNCTION subprogram is a kind of SLAVE subprogram consisting of an LFUNCTION statement followed by other statements including at least one RETURN statement. It is executed whenever its name is referred to in a global subprogram, or another local subprogram. When we call FUNCTION and LFUNCTION together we use a generic word '(L)FUNCTION'.

An (L)FUNCTION statement assigns the name to the (L)FUNCTION subprogram and gives a list of dummy argument names. It must be the first statement of the subprogram.

FUNCTION statement, LFUNCTION statement
[*Type*] FUNCTION *name* (*a* 1 , *a* 2 , . . . , *an*) [*Type*] LFUNCTION *name* (*a* 1 , *a* 2 , . . . , *an*)
Where : *Type* is a type declarator and it is optional. *name* is the name of the (L) FUNCTION. Each *a* is a dummy argument. It must be a disinct variable or array name or dummy procedure name. If there is no argument, the parenthesis must be present.

A FUNCTION statement may have dummy arguments of HOST variables, arrays, HOST procedures or global SLAVE subroutines. An LFUNCTION statement may have dummy arguments of SLAVE variables, or segments.

13.2.1 A type declaration for an (L)FUNCTION name may be made by the predefined convention, by an IMPLICIT statement, by an explicit specification in the (L)FUNCTION statement, or by an explicit specification statement within the

(L)FUNCTION subprogram.

The (L)FUNCTION name must also be typed in the program units which refer to it if the predefined convention is not used.

13.2.2 The (L) FUNCTION subprogram may contain any FORTRAN statement except a SUBROUTINE statement, another FUNCTION statement, BLOCKDATA statement, or a PROGRAM statement, but it must not contain any ADETRAN PROPER statement. The name of a function must not be in any other non-executable statement except a type statement.

13.2.3 The name of the function must be assigned a value at least once during the execution of the subprogram in one of the following ways:

- As the variable name to the left of the equal sign in an arithemetic, logical, or character (only permissible in the FUNCTION) assignment statement
- As an argument of a CALL statement that will cause a value to be assigned in the subroutine referred to
- As a DO- or implied DO-variable
- In the FUNCTION, there are other ways related with I/O statements

13.2.4 The value of the (L)FUNCTION is the last value assigned to the name of the function when a RETURN or END statement is executed in the subprogram. The (L)FUNCTION subprogram may also have one or more of its arguments to return values to the calling program. An argument so used must appear:

- On the left side of an arithmetic, logical, or character (only permissible in the FUNCTION) assignment statement
- As an argument in a FUNCTION reference that is assigned a value by the function referred to
- As an argument in a CALL statement that is assigned a value in the subroutine referred to
- In the FUNCTION, it may appear in another list or as a parameter in some statements related with I/O

13.2.5 If an (L)FUNCTION dummy argument is used as an adjustable array, the array name and all the variables in the array declarators (except those in the common block) must be in the dummy argument list.

13.2.6 Any (L)FUNCTION subprogram must contain an END statement and at least one RETURN statement. The END statement specifies the physical end of the subprogram; the RETURN statement signifies a logical conclusion of the computation and returns the computed function value and control to the calling program.

14. SUBROUTINE and LSUBROUTINE Subprogram

A SUBROUTINE subprogram is a kind of HOST subprograms, and it may be called by another HOST subprogram.

A LSUBROUTINE subprogram is a kind of SLAVE subprogram, and it may be called by a global SLAVE subprogram, or another local SLAVE subprogram.

When we mention them together, we use the word (L)SUBROUTINE subprogram, or simply (L)SUBROUTINE as the generic name. Like the (L) FUNCTION subprogram, the (L)SUBROUTINE subprogram is a set of commonly used computations, but it need not return any results to the calling program, as does the (L)FUNCTION subprogram. The (L)SUBROUTINE subprogram may contain any FUNCTION statements except a FUNCTION statement, another SUBROUTINE statement, or a PROGRAM statement, and it cannot contain any ADETRAN PROPER statement.

An (L)SUBROUTINE statement idetifies an (L)SUBROUTINE. It must be the first statement of the (L)SUBROUTINE.

SUBROUTINE and LSUBROUTINE statements
SUBROUTINE *name* (*a* 1 , *a* 2 , . . . , *an*) LSUBROUTINE *name* (*a* 1 , *a* 2 , . . . , *an*) Where : *name* is the (L)SUBROUTINE name. Each *a* is a distinct dummy argument. There need not be any arguments, in which case the parenthesis may be omitted.

14. .1 Each argument used in the SUBROUTINE statement must be a HOST variable, HOST array name, a dummy name of another SUBROUTINE or FUNCTION subprogram or an asterisk '*', where the character '*' denotes a return point specified by a statement number in the calling program.

Each argument used in the LSUBROUTINE statement must be a SLAVE variable, a segment or an asterisk '*', where the character '*' denotes a return point specified by a statement number in the calling global subprogram.

14. .2 The (L)SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. An argument so used will appear on the left side of an arithmetic, logical, or character (only permissible in the SUBROUTINE) assignment statement, or as an argument in a CALL statement or function reference that is assigned a value by the SUBROUTINE or

FUNCTION referred to, or in the list of a READ statement. The dummy arguments (*a1, a2, ..., an*) may be considered dummy names that are replaced at the time of execution by the actual arguments supplied in the CALL statement.

If an (L)SUBROUTINE dummy argument is used as an adjustable array name, the array name and all the variables in the array declarators (except those in the common block) must be in the dummy argument list.

15. Global subprogram

A global subprogram is most representative in the ADETRAN with respect to parallel processing, and it may only contain some parallel control statements of ADETRAN PROPER, in addition to FORTRAN statements.

There are two kinds of global subprograms : GSUBROUTINE and GSUBCOROUTINE subprogram.

15.1 GSUBROUTINE subprogram

A GSUBROUTINE subprogram is referenced by a CALL statement which is in a HOST program.

Though the GSUBROUTINE subprogram may also be considered as a set of commonly used computations as is the (L)FUNCTION or (L)SUBROUTINE subprogram, it is rather for controlling the whole SLAVE system, and, in this respect, it would be called the SLAVE main program.

The GSUBROUTINE subprogram may contain some parallel processing paragraphs, in each of which any sequence of FORTRAN statements is permissible, global DO statements, and slave system synchronizing statements (IFALL/IFANY).

The GSUBROUTINE cannot contain an (L)FUNCTION statement, (L) SUBROUTINE statement, another GSUBROUTINE statement, or I/O related statement. A GSUBROUTINE statement identifies a GSUBROUTINE subprogram. It must be the first statement of the GSUBROUTINE.

GSUBROUTINE statement
GSUBROUTINE *name* (*a*1, *a*2, ..., *an*) Where : *name* is the GSUBROUTINE name. Each *a* is a distinct dummy argument. There need not be any arguments, in which case the parenthesis may be

omitted.

15.1.1 Each argument used in the GSUBROUTINE statement must be a HOST array, a dummy name of a GSUBROUTINE, LSUBROUTINE or LFUNCTION subprogram or an astrerisk '*', where the character '*' denotes a return point specified by a statement number in the calling HOST program.

15.1.2 The GSUBROUTINE subprogram may use one or more of its arguments to return values to the calling HOST program. Arguments so used will appear on the left side of the PASS statement in the subprogram. The dummy arguments (*a1, a2, . . . an*) may be considered as dummy names that are replaced at the time of execution by the actual arguments supplied in the CALL statement of the HOST program.

15.2 GSUBCOROUTINE Subprogram

A GSUBCOROUTINE subprogram is executed when it is called by a GSUBROUTINE subprogram. It can only be called from outside of any parallel paragraphs. It cannot be called directly by a HOST program.

The GSUBCOROUTINE subprogram is similar to the GSUBROUTINE in some respects, but it has the following restrictions:

- No ability of data transfer to/from the HOST computer
- No permission of dummy arguments. Data pass to/from the calling GSUBROUTINE are only performed through the common data block
- No permission of parallel processing control statements except PDO statements.

The GSUBCOROUTINE cannot contain an (L)FUNCTION statement, (G/L) SUBROUTINE statement, another GSUBCOROUTINE statement, or I/O related statement.

A GSUBCOROUTINE statement identifies a GSUBCOROUTINE subprogram. It must be the first statement of the GSUBCOROUTINE.

GSUBCOROUTINE statement
GSUBCOROUTINE *name* Where : *name* is the GSUBCOROUTINE name.

16. CALL statement

A CALL statement is used to call a (G/L) SUBROUTINE or GSUBCOROUTINE subprogram. The CALL statement transfers control to the

called subprogram, and replaces the dummy arguments, if any, with the values of the actual arguments.

CALL statement
CALL *name* (*a* 1 , *a* 2 , . . . , *an*) Where : *name* is the name of the (L/G)SUBROUTINE or GSUBCOROUTINE subprogramor ENTRY point. The name may appear as a dummy procedure in an (L)FUNCTION, or (L/G)SUBROUTINE. Each *a* is an actual argument that is being supplied to the called subprogram.

The actual argument may be a variable, array element, array (segment) name, or a constant, an arithmetic, logical, or character (permissible only in HOST programs) expression, a subprogram name, or a statement number (preceded by an asterisk) of an executable statement in the same program unit as the call statement. In the case of no argument, the parenthesis may be omitted.

16. .1 The CALL statement can be used in a MAIN program, an (L) FUNCTION subprogram, or an (L/G)SUBROUTINE, but a subprogram must not refer to itself, directly or indirectly, and must not refer to the MAIN program. A MAIN program cannot call itself. A HOST program can call a (G)SUBROUTINE, but it cannot call a GSUBROUTINE or LSUBROUTINE directly.

A GSUBROUTINE can call a GSUBCOROUTINE or LSUBROUTINE, but it cannot call another GSUBROUTINE.

A GSUBCOROUTINE can call an LSUBROUTINE, but it cannot call another GSUBCOROUTINE or a GSUBROUTINE.

An LSUBROUTINE can call another LSUBROUTINE, but it cannot call any global subprogram.

16. .2 A CALL statement to an LSUBROUTINE in the GSUBROUTINE or GSUBCOROUTINE may have actual arguments of variables, non-directed segment or segments with a specific direction attribute.

Example :

(Calling GSUBROUTINE)	(LSUBROUTINE)
DIMENSION U(0 :33, / 32, 32 /)	LSUBROUTINE SW(V) DIMENSION V(0 :33)
⋮	⋮
PDO I, J = 1,32	

```
      CALL  SW(U(,/I, J/))        :
   PEND                           :
```

17. Other Subprogram Related Items

17.1 RETURN Statement

Execution of a RETURN statement transfers control to the calling program.

RETURN statement
RETURN RETURN *i* Where : *i* is an integer constant or variable expression whose value, say n, denotes the *n* th statement number in the parenthesized argument list of an (L/G) SUBROUTINE statement : *i* may be specified only in an (L/G)SUBROUTI NE.

The normal sequence of execution following the RETURN statement of an (L/G) SUBROUTINE or GSUBROUTINE is the next statement following CALL in the calling program.

The RETURN statement of an (L)FUNCTION subprogram returns the computed of function value and control to the calling program.

17.1.1 It is also possible to return to any number statement in the calling program by using a return of the type RETURN *i*, which may only be permissible in an (L/G)SUBROUTINE subprogram.

If $1 \leqq i \leqq m$, where m is the number of asterisks in the argument list of the (L/G) SUBROUTINE statement, the value of i, say n, specifies the n th asterisk in the dummy argument list. There should be a one to one correspondence between the number of alternate return specifiers specified in the CALL statement and the number of asterisks in the (L/G) SUBROUTINE statement dummy list. However, the alternate return specifiers need not be unique. Control is returned to the statement identified by the alternate return specifier in the CALL statement that is associated with the n th asterisk in the dummy argument list of the currently referenced name. This completes the execution of the CALL statement. If the value of i is less than one or greater than m, the control returns to the next statement following the CALL statement that initiated the subprogram reference.

17.2 Actual arguments

The actual arguments in a subroutine or function reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced (L/G)SUBROUTINE or FUNCTION. The use of an (L/G) SUBROUTINE, GSUBCOROUTINE or (L)FUNCTION name, or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type. An actual argument in a subroutine or function reference must be one of the following:

- An arithmetic, logical or character constant (character type is always valid only for calling a HOST subprogram)
- A variable, array element, or segment element
- An array name or segment name
- An intrinsic function name
- An external procedure name
- A dummy argument name that appears in a dummy argument list within the subprogram containing the reference.
- An arithmetic or logical expression
- An alternate return specifier (statement number preceded by an asterisk).

An actual argument which is the name of a subprogram must be identified by an EXTERN statement in the calling program unit containing that name. An actual argument which is the name of an intrinsic function must be identified by an INTRINSIC statement in the calling program unit containing that name.

17.3 EXTERNAL statement

An EXTERNAL statement identifies a user-supplied name and permits such a name to be used as an actual argument.

EXTERNAL statement
EXTERNAL *a* 1 , *a* 2 , . . . , *an* Where : Each *a* is the name of a user-supplied subprogram ((L)FUNCTION or (L/G)SUBROUTINE) that is passed as an argument to another subprogram

The EXTERNAL statement is a specification statement and must precede DATA statement, statement function definitions, and all executable statements. Statement function names cannot appear in EXTERNAL statements.

17.4 INTRINSIC statement

An INTRINSIC statement identifies a name as representing an ADETRAN-supplied prodedure (function or subprogram) and permits a specific intrinsic function name to be used as an actual argument. The INTRINSIC statement is a specification statement and must precede statement function definitions and all executable statements. Its format is the same as in FORTRAN.

17.5 Dummy arguments

The dummy arguments of a subprogram appear after the subprogram name and are enclosed in parentheses. They are replaced at the time of execution of the CALL statement by the actual arguments supplied in the CALL statement in the calling program.

17.5.1 Dummy arguments must follow certain rules:

- None of the dummy argument names may appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, or INTRINSIC statement except as common block names.
- A dummy argument name must not be the same as the entry name appearing in a PROGRAM, (L)FUNCTION, (L/G)SUBROUTINE, ENTRY, or statement function definition in the same program unit.
- The dummy arguments must correspond in number, order and type to the actual arguments. If a dummy argument is an array (segment), the corresponding actual argument must be either an array (segment) or an array (segment) element.
- IF a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, an array element, or an array (segment). A constant, constant name, subprogram name, or expression should not be written as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.
- The subprogram reserves no storage for the dummy argument, using the corresponding actual argument in the calling program for its calculations.
- A referenced subprogram cannot assign new values to dummy arguments that are associated with other dummy arguments within the subprogram or with variables in common.
- A dummy argument is an array when an appropriate DIMENSION or explicit type specification statement appears in the subprogram.

18. Other HOST statements

For comparison, we here list up other important statements permissible only in HOST subprograms and mention data initialization.

18.1 PROGRAM statement

A PROGRAM statement assigns a name to a main program. Its format is that of FORTRAN. The PROGRAM statement can only be used in a main program, but is not reqired. If it is not used, the name of the main program is assumed by the compiler to be MAIN.

18.2 ENTRY statement

The normal entry into a SUBROUTINE subprogram from the calling program is made by a CALL statement that refers to the subprogram name. The normal entry into a FUNCTION subprogram is made by a function reference in an expression. Entry is made at the first executable statement following the SUBROUTINE or FUNCTION statement. It is also possible to enter a SUBROUTINE or FUNCTION subprogram by a CALL statement or alternatively a function reference that references an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement.

In any SLAVE subprogram, only normal entry is permissible and no ENTRY statement may appear.

18.3 Input/Output statements

Input/Output is fully performed in Host programs, and Input/Output statements are the same as those in FORTRAN. The followings are fundamental:

1) READ statement
2) WRITE and PRINT statements

The followings are for processing external files:

3) OPEN statement
4) CLOSE statement
5) INQUIRE statement
6) ENDFILE statement
7) REWIND statement
8) BACKSPACE statement

The following is for indicating I/O formats:

9) FORMAT statement

18.4 Data initialization

A DATA statement defines the initial values of HOST variables or arrays in HOST subprograms.

A BLOCKDATA subprogram is a HOST subprogram, and it provides initial values for named common blocks of HOST data with using some DATA statements.

Any SLAVE data cannot be initialized without some assigment statements.

Reference

[1] T. Nogi, and M. Kubo, ADINA Computer I, I. Architecture and Theoretical Estimates, Memoirs of the Faculty of Engineering, Kyoto Universitey, 42(4) (1980) 421-439.

[2] T. Nogi, ADINA Computer II, I. Architecture and Theoretical Estimates, idid., 43(1) (1981) 124-144.

[3] T. Nogi, ADINA Computer I and II, II. DATA Structure, ibid., 43(3) (1981) 434-450.

[4] T. Nogi, Parallel Machine ADINA, in Computing Methods in Applied Sciences and Engineering, V, eds. R. Glowinsky and J. L. Lions, North-Holland, (1982) 103-122.

[5] T. Nogi, The ADENA Computer, in International Symposium on Applied Mathematics and Information Science, Kyoto University towards Multidimensional Flow Models, Mathematics and Computers, Kyoto University, (1984) 7/9-16.

[6] T. Nogi, Parallel Computation, in Patterns and Waves, Studies in Mathematics and its Applications 18, Kinokuniya/North-Holland, (1986) 279-318.

[7] IBM System/360 and System/370 FORTRAN IV Language (GC 28-6515-11)

[8] VS FORTRAN Language and Library Reference (SC 26-4119-1)

Index

actual argument ··· 285
ADE ··· 240
ADENA ··· 240
arithmetic expression ··· 257
arithmetic IF ··· 259
assigned GOTO ··· 259
assignment statement ··· 258
blank common block ··· 253
BLOCKDATA ··· 289
block IF ··· 261
block IFALL/IFANY ··· 276
character ··· 242
character value ··· 242
COMMON ··· 254
common block ··· 253
computed GOTO ··· 259
constant scalar ··· 245
constant vector ··· 245
CONTINUE ··· 263
DATA ··· 289
DIMENSION ··· 248
directed segment ··· 248
directed segment array ··· 247
DO ··· 262
DO paragraph ··· 263
dummy argument ··· 287
ELSE ··· 262
ELSEA ··· 277
ELSEIF ··· 262
ELSEIFALL/ELSEIFANY ··· 276
END ··· 264
ENDIF ··· 261
ENDIFA ··· 276
ENTRY ··· 288
EOF ··· 251
EQUIVALENCE ··· 255
explicit specification ··· 246
explicit type statement ··· 248
EXTERNAL ··· 286
FILE ··· 251
file array ··· 251
FUNCTION ··· 277
general control statement ··· 259
GET ··· 252
global subprogram ··· 244
GOTO ··· 259
GSUBCOROUTINE ··· 283
GSUBROUTINE ··· 282
HOST constant ··· 245
HOST data ··· 244
HOST expression ··· 258
HOST subprogram ··· 243
HOST variable ··· 245
IF ··· 259
IFALL/IFANY ··· 274
IF-block ··· 261
IMPLICIT ··· 246
INTRINSIC ··· 286
key word ··· 243
LFUNCTION ··· 279
local subprogram ··· 244
logical expression ··· 257
logical IF ··· 260
logical IFALL/IFANY ··· 274
logical value ··· 242
LSUBROUTINE ··· 280
name ··· 242
named common block ··· 253
non-directed segment ··· 247
non-directed segment array ··· 247
numerical value ··· 242
PAD ··· 272
parallel control statement ··· 264
parallel index region ··· 256
PARAMETER ··· 245
PASS ··· 266
PASS paragraph ··· 267
PAUSE ··· 264
PDO ··· 264
PDO paragraph ··· 264
PEND ··· 264, 267
PHASE ··· 268
pointwise sharing scheme ··· 238
POUR ··· 270
predefined specification ··· 246
processor number index ··· 248
PROGRAM ··· 288
program unit ··· 243
PUSH ··· 273
PUT ··· 253
RECORD ··· 251
record array ··· 251
REGION ··· 256
relational expression ··· 257
replacement ··· 267
RESET ··· 252
RETURN ··· 285
REWRITE ··· 252
segment ··· 239
segment array ··· 247
segment-parallel processing ··· 242
segmentwise sharing scheme ··· 238
SLAVE constant ··· 245
SLAVE data ··· 244
SLAVE expression ··· 258
SLAVE subprogram ··· 244
S-scheme ··· 241
statement function ··· 278
STOP ··· 264
SUBROUTINE ··· 281
transferred segment ··· 269
unconditional GOTO ··· 259
variable array ··· 246
variable scalar ··· 245
variable uni-vector ··· 245
vector ··· 242
vector-pipeline processing ··· 242
window ··· 251