Fortran 2018 Overview

December 21, 2022

1 Introduction

This document describes the new parts of the Fortran 2018 language that are supported by the latest release of the NAG Fortran Compiler.

The compiler release in which a feature was made available is indicated by square brackets; for example, a feature marked as '[5.3]' was first available in Release 5.3.

2 Overview of Fortran 2018

The new features of Fortran 2018 that are supported by the NAG Fortran Compiler can be grouped as follows:

- Data declaration
- Data usage and computation
- Input/output
- Execution control
- Intrinsic procedures and modules
- Program units and procedures
- Advanced C interoperability
- Updated IEEE arithmetic capabilities
- Advanced coarray programming

3 Data declaration

- [5.3] If an object is initialised (in a type declaration statement or component definition statement), its array bounds and character length can be used in its initialisation expression.
- [7.0] The `EQUIVALENCE` and `COMMON` statements, and the `BLOCK DATA` program unit, are considered to be obsolescent (and reported as such when the *–f2018* option is used).
- [7.1] Assumed-rank dummy arguments accept actual arguments of any rank; they **assume** the rank from the actual argument. This rank may be zero; that is, the actual argument may be scalar. Furthermore, assumed-rank dummy arguments may have the `ALLOCATABLE` or `POINTER` attribute, and thus accept allocatable/pointer variables of any rank.

  The syntax is as follows:

  ```
  Real,Dimension(..) :: a, b
  Integer :: c(..)
  ```

  That declares three variables (which must be dummy arguments) to be assumed-rank.

  The use of assumed-rank dummy arguments within Fortran is extremely limited; basically, the intrinsic inquiry functions can be used, and there is a `SELECT RANK` construct, but other than that they may only appear as actual arguments to other procedures where they correspond to another assumed-rank argument.

  The main use of assumed rank is for advanced C interoperability (see later section).

  Here is an extremely simple example of use within Fortran:

```
Program assumed_rank_example
  Real x(1,2),y(3,4,5,6,7)
  Call showrank(1.5)
  Call showrank(x)
  Call showrank(y)
Contains
  Subroutine showrank(a)
    Real,Intent(In) :: a(..)
    Print *,'Rank is',Rank(a)
  End Subroutine
End Program
```

That will produce the output

```
Rank is 0
Rank is 2
Rank is 5
```

- [7.1] The `TYPE(*)` type specifier can be used to declare scalar, assumed-size, and assumed-rank dummy arguments. Such an argument is called **assumed-type**; the corresponding actual argument may be of any type. It must not have the `ALLOCATABLE`, `CODIMENSION`, `INTENT (OUT)`, `POINTER`, or `VALUE` attribute.

  An assumed-type variable is extremely limited in the ways it can be used directly in Fortran:

  – it may be passed as an actual argument to another assumed-type dummy argument;
  – it may appear as the first argument to the intrinsic functions `IS_CONTIGUOUS`, `LBOUND`, `PRESENT`, `SHAPE`, `SIZE`, or `UBOUND`;
  – it may be used as the argument of the function `C_LOC` (in the `ISO_C_BINDING` intrinsic module}.

  Other than these contexts, it cannot be used in any other way at all. Note that if it is an array, you cannot subscript it or create an array section from it.

  This is mostly useful for interoperating with C programs (see later section). Note that in a non-generic procedure reference, a scalar argument can be passed to an assumed-type argument that is an assumed-size array.

4 Data usage and computation

- [7.1] The `SELECT RANK` construct facilitates use of assumed rank objects within Fortran. It has the syntax

```
[ construct-name ] SELECT RANK ( [ assoc_name => ] assumed-rank-variable-name )
      [ rank-stmt
          block ]...
END SELECT [ construct-name ]
```

  where *rank-stmt* is one of:

```
RANK ( scalar-int-constant-expression ) [ construct-name ]
RANK ( * ) [ construct-name ]
RANK DEFAULT [ construct-name ]
```

  In any particular `SELECT RANK` construct, there must not be more than one `RANK DEFAULT` statement, or more than one `RANK (*)` statement, or more than `RANK (`*integer*`)` with the same value integer expression. If the assumed-rank variable has the `ALLOCATABLE` or `POINTER` attribute, the `RANK (*)` statement is not permitted.

  The *block* following a `RANK` statement with an integer constant expression is executed if the assumed-rank variable is associated with a non-assumed-rank actual argument that has that rank, and is not an assumed-size array. Within the *block* it acts as if it were an assumed-shape array with that rank.

  The *block* following a `RANK (*)` is executed if the ultimate argument is an assumed-size array. Within the *block* it acts as if it were declared with bounds '`(1:*)`'; if different bounds or rank are desired, this can be passed to another procedure using sequence association.

The *block* following a `RANK DEFAULT` statement is executed if no other block is selected. Within its *block*, it is still an assumed-rank variable, i.e. there is no change.

Here is a simple example of the `SELECT RANK` construct.

```
Program select_rank_example
  Integer :: a = 123, b(1,2) = Reshape( [ 10,20 ], [ 1,2 ] ), c(1,3,1) = 777, d(1,1,1,1,1)
  Call show(a)
  Call show(b)
  Call show(c)
  Call show(d)
Contains
  Subroutine show(x)
    Integer x(..)
    Select Rank(x)
    Rank (0)
      Print 1,'scalar',x
    Rank (1)
      Print 1,'vector',x
    Rank (2)
      Print 1,'matrix',x
    Rank (3)
      Print 1,'3D array',x
    Rank Default
      Print *,'Rank',Rank(x),'not supported'
    End Select
  1 Format(1x,a,*(1x,i0,:))
  End Subroutine
End Program
```

This will produce the output

```
scalar 123
matrix 10 20
3D array 777 777 777
Rank 5 not supported
```

5 Input/output

- [7.0] The `RECL=` specifier in an `INQUIRE` statement for an unconnected unit or file now assigns the value -1 to the variable. For a unit or file connected with `ACCESS='STREAM'`, it assigns the value -2 to the variable. Under previous Fortran standards, the variable became undefined.
- [7.1] The `SIZE=` specifier can be used in a `READ` statement without `ADVANCE='NO'`, that is, in a `READ` statement with no `ADVANCE=` specifier, or one with an explicit `ADVANCE='YES'`. For example,

```
Character(65536) buf
Integer nc
Read(*,'(A)',Size=nc) buf
Print *,'The number of characters on that line was',nc
```

Note that `SIZE=` is not permitted with list-directed or namelist formatting; that would be pointless, as there are no edit descriptors with such formatting and thus no characters to be counted by `SIZE=`.

6 Execution control

- [6.2] The expression in an `ERROR STOP` or `STOP` statement can be non-constant. It is still required to be default Integer or default Character.

- [6.2] The `ERROR STOP` and `STOP` statements now have an optional `QUIET=` specifier, which is preceded by a comma following the optional stop-code. This takes a Logical expression; if it is true at runtime then the `STOP` (or `ERROR STOP`) does not output any message, and information about any IEEE exceptions that are signalling will be suppressed. For example,

```
STOP 13, QUIET = .True.
```

will not display the usual 'STOP: 13', but simply do normal termination, with a process exit status of 13. Note that this means that the following two statements are equivalent:

```
STOP, QUIET=.True.
STOP 'message not output', QUIET=.TRUE.
```

7 Intrinsic procedures and modules

- [6.2] The intrinsic subroutine `MOVE_ALLOC` now has optional `STAT` and `ERRMSG` arguments. The `STAT` argument must be of type Integer, with a decimal range of at least four (i.e. not an 8-bit integer); it is assigned the value zero if the subroutine executes successfully, and a nonzero value otherwise. The `ERRMSG` argument must be of type Character with default kind. If `STAT` is present and assigned a nonzero value, `ERRMSG` will be assigned an explanatory message (if it is present); otherwise, `ERRMSG` will retain its previous value (if any).

  For example,

```
INTEGER,ALLOCATABLE :: x(:),y(:)
INTEGER istat
CHARACTER(80) emsg
...
CALL MOVE_ALLOC(x,y,istat,emsg)
IF (istat/=0) THEN
  PRINT *,'Unexpected error in MOVE_ALLOC: ',TRIM(emsg)
```

  The purpose of these arguments is to catch errors in multiple image coarray allocation/deallocation, such as `STAT_STOPPED_IMAGE` and `STAT_FAILED_IMAGE`.

- [7.1] The `DIM` argument to the intrinsic functions `ALL`, `ANY`, `FINDLOC`, `IALL`, `IANY`, `IPARITY`, `MAXLOC`, `MAXVAL`, `MINLOC`, `MINVAL`, `NORM2`, `PARITY`, `PRODUCT` and `SUM` can be an optional dummy argument, as long as it is present at execution time. For example,

```
Subroutine sub(x,n)
  Real,Intent(In) :: x(:,:,:)
  Integer,Intent(In),Optional :: n
  If (Present(n)) Then
    Print *,Norm2(x,n) ! Rank two array result.
  Else
    Print *,Norm2(x)   ! Scalar result.
  End If
End Subroutine
```

- [7.1] Specific intrinsic functions are considered to be obsolescent (and reported as such with the *–f2018* option). In the case of a function that is both specific and generic, e.g. `SQRT`, the obsolescent usage is passing as an actual argument, use as a procedure interface, or being the target of a procedure pointer assignment.

- [7.1] The intrinsic inquiry function `RANK` returns the dimensionality of its argument. It has the following syntax:

```
RANK ( A )
```

  `A` : data object of any type:

Result : scalar Integer of default kind.

The result is the rank of `A`, that is, zero for scalar `A`, one if `A` is a one-dimensional array, and so on.

This function can be used in a constant expression except when `A` is an assumed-rank variable.

- [7.1] The intrinsic function `REDUCE` performs user-defined array reductions. It has the following syntax:

```
REDUCE ( ARRAY, OPERATION [, MASK, IDENTITY, ORDERED ] ) or
REDUCE ( ARRAY, OPERATION DIM [, MASK, IDENTITY, ORDERED ] )
```

`ARRAY` : array of any type;

`OPERATION` : pure function with two arguments, each argument being scalar, non-allocatable, non-pointer, non-polymorphic non-optional variables with the same declared type and type parameters as `ARRAY`; if one argument has the `ASYNCHRONOUS`, `TARGET` or `VALUE` attribute, the other must also have that attribute; the result must be a non-polymorphic scalar variable with the same type and type parameters as `ARRAY`;

`DIM` : scalar Integer in the range 1 to N, where N is the rank of `ARRAY`;

`MASK` : type Logical, and either scalar or an array with the same shape as `ARRAY`;

`IDENTITY` : scalar with the same declared type and type parameters as `ARRAY`;

`ORDERED` : scalar of type Logical;

Result : Same type and type parameters as `ARRAY`.

The result is `ARRAY` reduced by the user-supplied `OPERATION`. If `DIM` is absent, the whole (masked) `ARRAY` is reduced to a scalar result. If `DIM` is present, the result has rank N-1 and the shape of `ARRAY` with dimension `DIM` removed; each element of the result is the reduction of the masked elements in that dimension.

If exactly one element contributes to a result value, that value is equal to the element; that is, `OPERATION` is only invoked when more that one element appears.

If no elements contribute to a result value, the `IDENTITY` argument must be present, and that value is equal to `IDENTITY`.

For example,

```
Module triplet_m
  Type triplet
    Integer i,j,k
  End Type
Contains
  Pure Type(triplet) Function tadd(a,b)
    Type(triplet),Intent(In) :: a,b
    tadd%i = a%i + b%i
    tadd%j = a%j + b%j
    tadd%k = a%k + b%k
  End Function
End Module
Program reduce_example
  Use triplet_m
  Type(triplet) a(2,3)
  a = Reshape( [ triplet(1,2,3),triplet(1,2,4), &
                 triplet(2,2,5),triplet(2,2,6), &
                 triplet(3,2,7),triplet(3,2,8) ], [ 2,3 ] )
  Print 1, Reduce(a,tadd)
  Print 1, Reduce(a,tadd,1)
  Print 1, Reduce(a,tadd,a%i/=2)
```

```
  Print 1, Reduce(Array=a,Dim=2,Operation=tadd)
  Print 1, Reduce(a, Mask=a%i/=2, Dim=1, Operation=tadd, Identity=triplet(0,0,0))
1 Format(1x,6('triplet(',I0,',',I0,',',I0,')',:,'; '))
End Program
```

This will produce the output:

```
 triplet(12,12,33)
 triplet(2,4,7); triplet(4,4,11); triplet(6,4,15)
 triplet(8,8,22)
 triplet(6,6,15); triplet(6,6,18)
 triplet(2,4,7); triplet(0,0,0); triplet(6,4,15)
```

- [7.0] The intrinsic **atomic subroutines** ATOMIC_ADD, ATOMIC_AND, ATOMIC_CAS, ATOMIC_FETCH_ADD, ATOMIC_FETCH_AND, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR, ATOMIC_OR and ATOMIC_XOR are described under **Advanced coarray programming**.

- [7.1] The intrinsic **collective subroutines** CO_BROADCAST, CO_MAX, CO_MIN, CO_REDUCE and CO_SUM are described under **Advanced coarray programming**.

- [7.0] The intrinsic functions COSHAPE, EVENT_QUERY, FAILED_IMAGES, GET_TEAM, IMAGE_STATUS, STOPPED_IMAGES, and TEAM_NUMBER, and the changes to the intrinsic functions NUM_IMAGES and THIS_IMAGE, are described under **Advanced coarray programming**.

8 Program units and procedures

- [7.0] If a dummy argument of a function that is part of an OPERATOR generic has the VALUE attribute, it is no longer required to have the INTENT(IN) attribute.

  For example,

```
INTERFACE OPERATOR(+)
   MODULE PROCEDURE logplus
END INTERFACE
...
PURE LOGICAL FUNCTION logplus(a,b)
   LOGICAL,VALUE :: a,b
   logplus = a.OR.b
END FUNCTION
```

- [7.0] If the second argument of a subroutine that is part of an ASSIGNMENT generic has the VALUE attribute, it is no longer required to have the INTENT(IN) attribute.

  For example,

```
INTERFACE ASSIGNMENT(=)
   MODULE PROCEDURE asgnli
END INTERFACE
...
PURE SUBROUTINE asgnli(a,b)
   LOGICAL,INTENT(OUT) :: a
   INTEGER,VALUE :: b
   DO WHILE (IAND(b,NOT(1))/=0)
      b = IEOR(IAND(b,1),SHIFTR(b,1))
   END DO
   a = b/=0 ! Odd number of "1" bits.
END SUBROUTINE
```

- [7.0] With the *–recursive* or the *–f2018* option, procedures are recursive by default. For example, this subprogram

```
INTEGER FUNCTION factorial(n) RESULT(r)
   IF (n>1) THEN
      r = n*factorial(n-1)
   ELSE
      r = 1
   END IF
END FUNCTION
```

is valid, just as if it had been explicitly declared with the `RECURSIVE` keyword.

This does not apply to assumed-length character functions (where the result is declared with `CHARACTER(LEN=*)`; these remain prohibited from being declared `RECURSIVE`.

Note that procedures that are `RECURSIVE` by default are excluded from the effects of the *−save* option, exactly as if they were explicitly declared `RECURSIVE`.

- [7.0] Elemental procedures may now be recursive, whether explicitly declared `RECURSIVE` or by default (when the *−f2018* or *−recursive* options are specified). For example,

```
ELEMENTAL RECURSIVE INTEGER FUNCTION factorial(n) RESULT(r)
   INTEGER,INTENT(IN) :: n
   IF (n>1) THEN
      r = n*factorial(n-1)
   ELSE
      r = 1
   END IF
END FUNCTION
```

may be invoked with

```
PRINT *,factorial( [ 1,2,3,4,5 ] )
```

to print the first five factorials.

- The `NON_RECURSIVE` keyword explicitly declares that a procedure will not be called recursively. For example,

```
NON_RECURSIVE INTEGER FUNCTION factorial(n) RESULT(r)
   r = 1
   DO i=2,n
      r = r*i
   END DO
END FUNCTION
```

In Fortran 2008 and older standards, procedures are non-recursive by default, so this keyword has no effect unless the *−recursive* or *−f2018* is being used.

- Generic resolution can use the number of procedure arguments; that is, if one procedure has more non-optional procedure arguments than the other has optional plus non-optional procedure arguments, the procedures are considered to be unambiguous.

For example,

```
MODULE npa_example
  INTERFACE g
    MODULE PROCEDURE s1,s2
  END INTERFACE
CONTAINS
  SUBROUTINE s1(a)
    EXTERNAL a
    CALL a
  END SUBROUTINE
  SUBROUTINE s2(b,a)
    EXTERNAL b,a
```

```
    CALL b
    CALL a
  END SUBROUTINE
END MODULE
```

This example does not conform to the Fortran 2008 rules for unambiguous generic procedures, because the argument `A` distinguishes by position but not by keyword, the argument `B` distinguish by keyword but not by position, and the positional disambiguator (`A`) does not appear earlier in the list than the keyword disambiguator (`B`).

9 Advanced C interoperability

- [7.0] The `C_FUNLOC` function from the intrinsic module `ISO_C_BINDING` accepts a non-interoperable procedure argument. The `C_FUNPTR` value produced should not be converted to a C function pointer, but may be converted to a suitable (also non-interoperable) Fortran procedure pointer with the `C_F_PROCPOINTER` subroutine from `ISO_C_BINDING`. For example,

```
USE ISO_C_BINDING
ABSTRACT INTERFACE
  SUBROUTINE my_callback_interface(arg)
    CLASS(*) arg
  END SUBROUTINE
END INTERFACE
TYPE,BIND(C) :: mycallback
  TYPE(C_FUNPTR) :: callback
END TYPE
...
TYPE(mycallback) cb
PROCEDURE(my_callback_interface),EXTERNAL :: sub
cb%callback = C_FUNLOC(sub)
...
PROCEDURE(my_callback_interface),POINTER :: pp
CALL C_F_PROCPOINTER(cb%callback,pp)
CALL pp(...)
```

  This functionality may be useful in a mixed-language program when the C_FUNPTR value is being stored in a data structure that is manipulated by C code.

- [7.0] The `C_LOC` function from the intrinsic module `ISO_C_BINDING` accepts an array of non-interoperable type, and the `C_F_POINTER` function accepts an array pointer of non-interoperable type. The array must still be non-polymorphic and contiguous.

  This improves interoperability with mixed-language C and Fortran programming, by letting the program pass an opaque "handle" for a non-interoperable array through a C routine or C data structure, and reconstruct the Fortran array pointer later. This kind of usage was previously only possible for scalars.

- [7.1] Assumed-rank variables are permitted to be dummy arguments of a `BIND(C)` routine, even those with the `ALLOCATABLE` or `POINTER` attribute. An assumed-rank argument is passed by reference as a "C descriptor"; it is then up to the C routine to decode what that means. The C descriptor, along with several utility functions for manipulating it, is defined by the source file `ISO_Fortran_binding.h`; this can be found in the compiler's library directory (on Linux this is usually `/usr/local/lib/NAG_Fortran`, but that can be changed at installation time).

  This topic is highly complex, and beyond the scope of this document. The reader should direct their attention to the Fortran 2018 standard, or to a good textbook.

- [7.1] A `TYPE(*)` ("assumed type") dummy argument is permitted in a `BIND(C)` procedure. It interoperates with a C argument declared as "`void *`". There is no difference between scalar and assumed-size on the C side, but on the Fortran side, if the dummy argument is scalar the actual argument must also be scalar, and if the dummy argument is an array, the actual argument must also be an array.

Because an actual argument can be passed directly to a `TYPE(*)` dummy, the `C_LOC` function is not required, and so there is no need for the `TARGET` attribute on the actual argument.

For example,

```
Program type_star_example
  Interface
    Function checksum(scalar,size) Bind(C)
      Use Iso_C_Binding
      Type(*) scalar
      Integer(C_int),Value :: size
      Integer(C_int) checksum
    End Function
  End Interface
  Type myvec3
    Double Precision v(3)
  End Type
  Type(myvec3) x
  Call Random_Number(x%v)
  Print *,checksum(x,Storage_Size(x)/8)
End Program
int checksum(void *a,int n)
{
  int i;
  int res = 0;
  unsigned char *p = a;
  for (i=0; i<n; i++) res = 0x3fffffff&((res<<1) + p[i]);
  return res;
}
```

- A `BIND(C)` procedure can have optional arguments. Such arguments cannot also have the `VALUE` attribute.

  An absent optional argument of a `BIND(C)` procedure is indicated by passing a null pointer argument.

  For example,

```
Program optional_example
  Use Iso_C_Binding
  Interface
    Function f(a,b) Bind(C)
      Import
      Integer(C_int),Intent(In) :: a
      Integer(C_int),Intent(In),Optional :: b
      Integer(C_int) f
    End Function
  End Interface
  Integer(C_int) x,y
  x = f(3,14)
  y = f(23)
  Print *,x,y
End Program

int f(int *arg1,int *arg2)
{
  int res = *arg1;
  if (arg2) res += *arg2;
  return res;
}
```

  The second reference to `f` is missing the optional argument `b`, so a null pointer will be passed for it. This will result in the output:

```
 17 23
```

10 Updated IEEE arithmetic capabilities

- [7.0] The module IEEE_ARITHMETIC has new functions IEEE_NEXT_DOWN and IEEE_NEXT_UP. These are elemental with a single argument, which must be a REAL of an IEEE kind (that is, IEEE_SUPPORT_DATATYPE must return .TRUE. for that kind of REAL). They return the next IEEE value, that does not compare equal to the argument, in the downwards and upwards directions respectively, except that the next down from $-\infty$ is $-\infty$ itself, and the next up from $+\infty$ is $+\infty$ itself. These functions are superior to the old IEEE_NEXT_AFTER function in that they do not signal any exception unless the argument is a signalling NaN (in which case IEEE_INVALID is signalled).

  For example, IEEE_NEXT_UP(-0.0) and IEEE_NEXT_UP(+0.0) both return the smallest positive subnormal value (provided subnormal values are supported), without signalling IEEE_UNDERFLOW (which IEEE_NEXT_AFTER does).

  Similarly, IEEE_NEXT_UP(HUGE(0.0)) returns $+\infty$ without signalling overflow.

- [7.0] The module IEEE_ARITHMETIC has new named constants IEEE_NEGATIVE_SUBNORMAL, IEEE_POSITIVE_SUBNORMAL, and the new function IEEE_SUPPORT_SUBNORMAL. These are from Fortran 2018, and reflect the change of terminology in the IEEE arithmetic standard in 2008. They are equivalent to the old functions IEEE_NEGATIVE_DENORMAL, IEEE_POSITIVE_DENORMAL and IEEE_SUPPORT_DENORMAL.

- [7.0] The requirement that the FLAG_VALUE argument to IEEE_GET_FLAG and IEEE_SET_FLAG, the HALTING argument to IEEE_GET_HALTING_MODE and IEEE_SET_HALTING_MODE, and the GRADUAL argument to IEEE_GET_UNDERFLOW_MODE and IEEE_SET_UNDERFLOW_MODE, be default LOGICAL has been dropped; any kind of LOGICAL is now permitted.

  For example,

```
USE F90_KIND
USE IEEE_ARITHMETIC
LOGICAL(byte) flags(SIZE(IEEE_ALL))
CALL IEEE_GET_FLAG(IEEE_ALL,flags)
```

  will retrieve the current IEEE flags into an array of one-byte LOGICALs.

11 Advanced coarray programming

- [7.0] Additional intrinsic atomic subroutines provide a means for multiple images to update atomic variables without synchronisation. These are:

```
      ATOMIC_ADD  (ATOM,VALUE,STAT)
      ATOMIC_AND  (ATOM,VALUE,STAT)
      ATOMIC_CAS  (ATOM,OLD,COMPARE,NEW,STAT)
ATOMIC_FETCH_ADD  (ATOM,VALUE,OLD,STAT)
ATOMIC_FETCH_AND  (ATOM,VALUE,OLD,STAT)
 ATOMIC_FETCH_OR  (ATOM,VALUE,OLD,STAT)
ATOMIC_FETCH_XOR  (ATOM,VALUE,OLD,STAT)
       ATOMIC_OR  (ATOM,VALUE,STAT)
      ATOMIC_XOR  (ATOM,VALUE,STAT)
```

  The arguments ATOM, COMPARE, NEW and OLD are all INTEGER(ATOMIC_INT_KIND). The ATOM argument is the one that is updated, and must be a coarray or a coindexed variable. The OLD argument is INTENT(OUT), and receives the value of ATOM before the operation. The STAT argument is optional, and must be a non-coindexed variable of type INTEGER and at least 16 bits in size.

  The VALUE argument must be INTEGER but can be of any kind; however, both VALUE and the result of the operation must be representable in INTEGER(ATOMIC_INT_KIND).

  The *_ADD operation is addition, the *_AND operation is bitwise and (like IAND), the *_OR operation is bitwise or (like IOR) and the *_XOR operation is bitwise exclusive or (like IEOR).

  ATOMIC_CAS is an atomic compare-and-swap operation. If ATOM is equal to COMPARE, it is assigned the value NEW; otherwise, it remains unchanged. In either case, the value before the operation is assigned to OLD. Note that both COMPARE and NEW must also be INTEGER(ATOMIC_INT_KIND).

  If the ATOM is a coindexed variable, and is located on a failed image, the operation fails and an error condition is raised; the OLD argument becomes undefined, and if STAT is present, it is assigned the value STAT_FAILED_IMAGE;

if STAT is not present, the program is terminated. If no error occurs and STAT is present, it is assigned the value zero.

- [7.0] The intrinsic function COSHAPE returns a vector of the co-extents of a coarray; its syntax is as follows.

  COSHAPE(COARRAY *[*, KIND *]*)

  COARRAY : coarray of any type; if it is ALLOCATABLE, it must be allocated; if it is a structure component, the rightmost component must be a coarray component;

  KIND (optional) : scalar Integer constant expression that is a valid Integer kind number;

  Result : vector of type Integer, or Integer(KIND)} if KIND is present; the size of the result is equal to the co-rank of COARRAY.

  For example, if a coarray is declared

  ```
  REAL x[5,*]
  ```

  and there are eight images in the current team, COSHAPE(x) will be equal to [5,2].

- [7.0] The intrinsic elemental function IMAGE_STATUS enquires whether another image has stopped or failed; its syntax is as follows.

  IMAGE_STATUS(IMAGE *[*, TEAM *]*)

  IMAGE : positive integer that is a valid image number;

  TEAM (optional) : scalar TEAM_TYPE value that identifies the current or an ancestor team;

  Result : default Integer.

  The value of the result is STAT_FAILED_IMAGE if the image has failed, STAT_STOPPED_IMAGE if the image has stopped, and zero otherwise. The optional TEAM argument specifies which team the image number applies to; if it is not specified, the current team is used.

- [7.0] The intrinsic function STOPPED_IMAGES returns an array listing the images that have initiated normal termination (i.e. "stopped"); its syntax is as follows.

  STOPPED_IMAGES(*[* TEAM, KIND *]*)

  TEAM (optional) : scalar TEAM_TYPE value that identifies the current or an ancestor team;

  KIND (optional) : scalar INTEGER constant expression that is a valid Integer kind number;

  Result : vector of type Integer, or Integer(KIND) if KIND is present.

  The elements of the result are the stopped image numbers in ascending order.

- [7.0] The type EVENT_TYPE in the intrinsic module ISO_FORTRAN_ENV, along with new statements and the intrinsic function EVENT_QUERY, provides support for **events**, a lightweight one-sided synchronisation mechanism.

  Like type LOCK_TYPE, entities of type EVENT_TYPE are required to be variables or components, variables of type EVENT_TYPE are required to be coarrays, and variables with noncoarray subcomponents of type LOCK_TYPE are required to be coarrays. Such variables are called **event variables**. An event variable is not permitted to appear in a variable definition context (i.e. any context where it might be modified), except in an EVENT POST or EVENT WAIT statement, or as an actual argument where the dummy argument is INTENT(INOUT).

  An event variable on an image may have an event "posted" to it by means of the image control statement EVENT POST, which has the form

  EVENT POST (*event-variable [, sync-stat]...*)

where the optional *sync-stat*s may be a single `STAT=`*stat-variable* specifier and/or a single `ERRSMG=`*errmsg-variable* specifier; *stat-variable* must be a scalar integer variable that can hold values up to 9999, and *errmsg-variable* must be a scalar default character variable. Posting an event increments the variable's "outstanding event count" (this count is initially zero). The *event-variable* in this statement will usually be a coindexed variable, as it is rarely useful for an image to post an event to itself.

If `STAT=` appears and the post is successful, zero is assigned to the *stat-variable*. If the image on which the *event-variable* is located has stopped, `STAT_STOPPED_IMAGE` is assigned to the *stat-variable*; if the image has failed, `STAT_FAILED_IMAGE` is assigned, and if any other error occurs, some other positive value is assigned. If `ERRMSG=` appears and any error occurs, an explanatory message is assigned to the *errmsg-variable*. Note that if `STAT=` does not appear and an error occurs, the program will be error-terminated, so having `ERRMSG=` without `STAT=` is useless.

Events are received by the image control statement `EVENT WAIT`, which has the form

```
EVENT WAIT ( event-variable [, event-wait-spec-list ] )
```

where the optional *event-wait-spec-list* is a comma-separated list that may contain a single `STAT=`*stat-variable* specifier, a single `ERRSMG=`*errmsg-variable* specifier, and/or a single `UNTIL_COUNT=`*scalar-integer-expr* specifier. Waiting on an event waits until its "outstanding event count" is greater than or equal to the `UNTIL_COUNT=` specifier value, or greater than zero if `UNTIL_COUNT=` does not appear. If the value specified in `UNTIL_COUNT=` is less than one, it is treated as if it were equal to one.

The *event-variable* in this statement is not permitted to be coindexed; that is, an image can only wait for events posted to its own event variables. There is a partial synchronisation between the waiting image and the images that contributed to the "outstanding event count"; the segment following execution of the `EVENT WAIT` statement follows the segments before the `EVENT POST` statement executions. The synchronisation does not operate in reverse, that is, there is no implication that execution of any segment in a posting image follows any segment in the waiting image.

The `STAT=` and `ERRMSG=` operate similarly to the `EVENT POST` statement, except of course that `STAT_FAILED_IMAGE` and `STAT_STOPPED_IMAGE` are impossible.

Finally, the intrinsic function `EVENT_QUERY` can be used to interrogate an event variable without waiting for it. It has the form

```
EVENT_QUERY ( EVENT, COUNT [, STAT ] )
```

where `EVENT` is an event variable, `COUNT` is an integer variable at least as big as default integer, and the optional `STAT` is an integer variable that can hold values up to 9999. `EVENT` is not permitted to be a coindexed variable; that is, only the image where the event variable is located is permitted to query its count. `COUNT` is assigned the current "outstanding event count" of the event variable. If `STAT` is present, it is assigned the value zero on successful execution, and a positive value if any error occurs. If any error occurs and `STAT` is not present, the program is error-terminated.

Note that event posts in unordered segments might not be included in the value assigned to count; that is, it might take some (communication) time for an event post to reach the variable, and it is only guaranteed to have reached the variable if the images have already synchronised. Use of `EVENT_QUERY` does not by itself imply any synchronisation.

- [7.0] The type `TEAM_TYPE` in the intrinsic module `ISO_FORTRAN_ENV`, along with new statements and intrinsic procedures, provides support for **teams**, a new method of structuring coarray parallel computation. The basic idea is that while executing inside a team, the coarray environment acts as if only the images in the team exist. This facilitates splitting coarray computations into independent parts, without the hassle of passing around arrays listing the images that are involved in a particular part of the computation.

  Unlike `EVENT_TYPE` and `LOCK_TYPE`, functions that return `TEAM_TYPE` are permitted. Furthermore, a variable of type `TEAM_TYPE` is forbidden from being a coarray, and assigning a `TEAM_TYPE` value from another image (e.g. as a component of a derived type assignment) makes the variable undefined; this is because the `TEAM_TYPE` value might contain information specific to a particular image, e.g. routing information to the other images. Variables of type `TEAM_TYPE` are called **team variables**.

  Creating teams

  The set of all the images in the program is called the **initial team**. At any time, a particular image will be executing in a particular team, the **current team**. A set of subteams of the current team can be created at any time by using the `FORM TEAM` statement, which has the form

```
FORM TEAM ( team-number , team-variable [, form-team-spec ]... )
```

where *team-number* is a scalar integer expression that evaluates to a positive value, *team-variable* is a team variable, and each *form-team-spec* is `STAT=`, `ERRMSG=`, and `NEW_INDEX=`*index-value* specifier. At most one of each kind of *form-team-spec* may appear in a `FORM TEAM` statement. All active images of the current team must execute the same `FORM TEAM` statement. If `NEW_INDEX=` appears, *index-value* must be a positive scalar integer (see below). The `STAT=` and `ERRMSG=` specifiers have their usual form and semantics.

The number of subteams that execution of `FORM TEAM` produces is equal to the number of unique *team-number* values in that execution; each unique *team-number* value identifies a subteam in the set, and each image belongs to the subteam whose team number it specified. If `NEW_INDEX=` appears, it specifies the image number that the image will have in its new subteam, and therefore must be in the range 1 to N, where N is the number of images in that subteam, and must be unique. If `NEW_INDEX=` does not appear, it is processor-dependent what the image number in the new subteam will be.

For example,

```
TYPE(TEAM_TYPE) oddeven
myteamnumber = 111*(MOD(THIS_IMAGE(),2) + 1)
FORM TEAM ( myteamnumber, oddeven )
```

will create a set of two subteams, one with team number 111, the other with team number 222. Team 111 will contain the images with even image numbers in the current team, and team 222 will contain the images with odd image numbers in the current team. On each image, the variable `oddeven` identifies the subteam to which that image belongs.

Note that the team numbers are completely arbitrary (being chosen by the program), and only have meaning within that set of subteams, which are called "sibling" teams.

Changing to a subteam

The current team is changed by executing a `CHANGE TEAM` construct, which has the basic form:

```
CHANGE TEAM ( team-value [, sync-stat-list ] )
  statements
END TEAM [ ( [  sync-stat-list  ] ) ]
```

where *team-value* is a value of type `TEAM_TYPE`, and the optional *sync-stat-list* is a comma-separated list containing at most one `STAT=` and `ERRMSG=` specifier; the `STAT=` and `ERRMSG=` specifiers have their usual form and semantics. Execution of the *statements* within the construct are with the current team set to the team identified by *team-value*; this must be a subteam of the current team outside the construct. The setting of the current team remains in effect during procedure calls, so any procedure referenced by the construct will also be executed with the new team current.

Transfer of control out of the construct, e.g. by a `RETURN` or `GOTO` statement is prohibited. The construct may be exited by executing its `END TEAM` statement, or by executing an `EXIT` statement that belongs to the construct; the latter is only possible if the construct is given a name (this is not shown in the form above, but consists of "*construct-name*`:`" prefix to the `CHANGE TEAM` statement, and and a "*construct-name*" suffix to the `END TEAM` statement).

While executing a `CHANGE TEAM` construct, image selectors operate using the new team's image indices, the intrinsic functions `NUM_IMAGES` and `THIS_IMAGES` return the data for the new team, and `SYNC ALL` synchronises the new team only.

There is an implicit synchronisation of all images of the new team both on the `CHANGE TEAM` statement, and on the `END TEAM` statement, and all active images must execute the same statement at this time.

Synchronising parent or ancestor teams

While executing within a `CHANGE TEAM` construct, the effects of `SYNC ALL` and `SYNC IMAGES` only apply to images within the current team. For `SYNC ALL` to synchronise the parent team, it would be necessary to first exit the construct. This may be inconvenient when the computation following the synchronisation would be within the team.

For this purpose, the `SYNC TEAM` statement has been added, with the form

```
SYNC TEAM ( team-value [, sync-stat-list ] )
```

where *team-value* identifies the current team or an ancestor thereof, and *sync-stat-list* is the usual comma-separated list containing at most one `STAT=` specifier and at most one `ERRMSG=` specifier (these have their usual semantics and so are not further described here).

The effect is to synchronise all images in the specified team.

Team-related intrinsic functions

- The intrinsic function GET_TEAM returns a value of type TEAM_TYPE that identifies a particular team. (This is the only way to get a TEAM_TYPE value for the initial team.) The function has the form

  ```
  GET_TEAM( [ LEVEL ] )
  ```

  where the optional LEVEL argument is a scalar integer value that is equal to one of the named constants CURRENT_TEAM, INITIAL_TEAM or PARENT_TEAM, in the intrinsic module ISO_FORTRAN_ENV. This argument specifies which team the returned TEAM_TYPE value should identify; if it is absent, the value for the current team is returned. If the current team is the initial team, the LEVEL argument must not be equal to PARENT_TEAM, as the initial team has no parent.
- The intrinsic function TEAM_NUMBER returns a team number value (that was used in FORM TEAM by the executing image). It has the form

  ```
  TEAM_NUMBER( [ TEAM ] )
  ```

  where the optional TEAM argument specifies which team to return the information for; it must identify the current team or an ancestor team, not a subteam or unrelated team. If TEAM is absent, the team number for the current team is returned. The initial team is considered to have a team number of -1 (except for the initial team, all team numbers are positive values).

Information about sibling and ancestor teams

The intrinsic functions NUM_IMAGES and THIS_IMAGE normally return information relevant to the current team, but they can return information for an ancestor team by using the optional TEAM argument, which takes a TEAM_TYPE value that identifies the current team or an ancestor. Similarly, the NUM_IMAGES intrinsic can return information for a sibling team by using the optional TEAM_NUMBER argument, which takes an integer value that is equal to the team number of the current or a sibling team. (Note that because the executing image is never a member of a sibling team, THIS_IMAGE does not accept a TEAM_NUMBER argument.) The intrinsic function NUM_IMAGES thus has two additional forms as follows:

```
NUM_IMAGES( TEAM )
NUM_IMAGES( TEAM_NUMBER )
```

For THIS_IMAGE, the revised forms it may take are as follows:

```
THIS_IMAGE( [ TEAM ] )
THIS_IMAGE( COARRAY [, TEAM ] )
THIS_IMAGE( COARRAY, DIM [, TEAM ] )
```

The meanings of the COARRAY and DIM arguments is unchanged. The optional TEAM argument specifies the team for which to return the information.

Establishing coarrays

A coarray is not allowed to be used within a team unless it is **established** in that team or an ancestor thereof. The basic rules for establishment are as follows:

1. a nonallocatable coarray with the SAVE attribute (explicit or implicit) is always established;
2. an unallocated coarray (with the ALLOCATABLE attribute) is not established;
3. an allocated coarray is established in the team where it was allocated;
4. a dummy coarray is established in the team that executed the procedure call (this may be different from the team where the actual argument is established).

Allocating and deallocating coarrays in teams

If a coarray with the ALLOCATABLE attribute is already allocated when a CHANGE TEAM statement is executed, it is not allowed to DEALLOCATE it within that construct (or within a procedure called from that construct).

If a coarray with the ALLOCATABLE attribute is unallocated when a CHANGE TEAM statement is executed, it may be allocated (using ALLOCATE) within that construct (or within a procedure called from that construct), and may be subsequently deallocated as well. If such a coarray remains allocated when the END TEAM statement is executed, it is automatically deallocated at that time.

This means that when using teams, allocatable coarrays may be allocated on some images (within the team), but unallocated on other images (outside the team), or allocated with a different shape or type parameters on other images (also outside the team). However, when executing in a team, the coarray is either unallocated on all images of the team, or allocated with the same type parameters and shape on all images of the team.

Accessing coarrays in sibling teams

Access to a coarray outside the current team, but in a sibling team, is possible using the TEAM_NUMBER= specifier in an image selector. This uses the extended syntax for image selectors:

```
[ cosubscript-list [, image-selector-spec-list ] ]
```

where *cosubscript-list* is the usual list of cosubscripts, and *image-selector-spec-list* contains a `TEAM_NUMBER=`*team-number* specifier, where *team-number* is the positive integer value that identifies a sibling team. The *image-selector-spec-list* may also contain a `STAT=` specifier (this is described later, under Fault tolerance).

When the `TEAM_NUMBER=` specifier is used the cosubscripts are treated as cosubscripts in the sibling team. Note that access in this way is quite risky, and will typically require synchronisation, possibly of the whole parent team. The coarray in question must be *established* in the parent team.

Accessing coarrays in ancestor teams

Access to a coarray in the parent or more distant ancestor team is possible using the `TEAM=` specifier in an image selector. This uses the extended syntax for image selectors:

```
[ cosubscript-list [, image-selector-spec-list ] ]
```

where *cosubscript-list* is the usual list of cosubscripts, and *image-selector-spec-list* contains a `TEAM=`*team-value* specifier, where *team-value* is a value of type `TEAM_TYPE` that identifies the current team or an ancestor. The *image-selector-spec-list* may also contain a `STAT=` specifier (this is described later, under Fault tolerance).

When the `TEAM=` specifier is used the cosubscripts are treated as cosubscripts in the specified ancestor team, and the image thus specified may lie within or outside the current team. If the access is to an image that is outside the current team, care should be taken that the images are appropriately synchronised; such synchronisation cannot be obtained by `SYNC ALL` or `SYNC IMAGES`, as they operate within a team, but may be obtained by `SYNC TEAM` specifying an ancestor team, or by using locks or events. The coarray in question must be *established* in the specified (current or ancestor) team.

Coarray association in `CHANGE TEAM`

It is possible to associate a local *coarray-name* in a `CHANGE TEAM` construct with a named coarray outside the construct, changing the codimension and/or coextents in the process. This acts like a limited kind of argument association; the local *coarray-name* has the type, parameters, rank and array shape of the outside coarray, but does not have the `ALLOCATABLE` attribute. The syntax of the `CHANGE TEAM` construct with one or more such associations is as follows:

```
CHANGE TEAM ( team-value , coarray-association-list [, sync-stat-list ] )
```

where *coarray-association-list* is a comma-separated list of

```
local-coarray-name [ explicit-coshape-spec ] => outer-coarray-name
```

and *explicit-coshape-spec* is

```
[ [ lower-cobound : ] upper-cobound , ]... [ lower-cobound : ] *
```

(The notation *[something]...* means *something* occurring zero or more times.)

The cobounds expressions are evaluated on execution of the `CHANGE TEAM` statement.

Use of this feature is not encouraged, as it is less powerful and more confusing than argument association.

- [7.0] Fault tolerance features for coarrays are supported. These consist of the `FAIL IMAGE` statement, the named constant `STAT_FAILED_IMAGE` in the intrinsic module `ISO_FORTRAN_ENV`, the `STAT=` specifier in an image selector, and the intrinsic function `FAILED_IMAGES`.

  The form of the `FAIL IMAGE` statement is simply

  ```
  FAIL IMAGE
  ```

  and execution of this statement will cause the current image to "fail", that is, cease to participate in program execution. This is the only way that an image can fail in NAG Fortran 7.0.

  If all images have failed or stopped, program execution will terminate. NAG Fortran will display a warning message if any images have failed.

  An image selector has an optional list of specifiers, the revised syntax of an image selector being (where the normal square brackets are literally square brackets, and the italic square brackets indicate optionality):

  ```
  [ cosubscript-list [, image-selector-spec-list ] ]
  ```

  where *cosubscript-list* is a comma-separated list of cosubscripts, one scalar integer per codimension of the variable, and *image-selector-spec-list* is a comma-separated containing at most one `STAT=`*stat-variable* specifier, and at most one `TEAM=` or `TEAM_NUMBER=` specifier (these were described earlier). If the coindexed object being accessed

lies on a failed image, the value STAT_FAILED_IMAGE is assigned to the *stat-variable*, and otherwise the value zero is assigned.

The intrinsic function FAILED_IMAGES returns an array of images that are known to have failed (it is possible that an image might fail and no other image realise until it tries to synchronise with it). Its syntax is as follows.

FAILED_IMAGES(*[* TEAM, KIND *]*)

TEAM (optional) : scalar TEAM_TYPE value that identifies the current or an ancestor team;

KIND (optional) : scalar Integer constant expression that is a valid Integer kind number;

Result : vector of type Integer, or Integer(KIND) if KIND is present.

The elements of the result are the failed image numbers in ascending order.

In order to be able to handle failed images, the following semantics apply:

- writing a value to a variable on a failed image is permitted (but may have no effect);
- reading a value from a variable on a failed image is permitted, but the result is unpredictable;
- execution of a CHANGE TEAM, END TEAM, FORM TEAM, SYNC ALL, SYNC IMAGES or SYNC TEAM statement with a STAT= specifier is permitted, and performs the team change, creation, or synchronisation operation on the non-failed images, assigning the value STAT_FAILED_IMAGE to the STAT= variable.

The latter effect in particular allows the program to form a team of all the non-failed images, and keep executing normally. However, since the data on the failed images is lost (reading the data produces garbage), the program would need to be carefully designed to "checkpoint" its work periodically, so that it can roll the computation state back to a known good value to recover.

The fault tolerance features are in principle intended to permit recovery from hardware failure, with the FAIL IMAGE statement allowing some testing of recovery scenarios. The NAG Fortran Compiler does not support recovery from hardware failure (at Release 7.1).

- [7.1] The intrinsic subroutines CO_BROADCAST, CO_MAX, CO_MIN, CO_REDUCE and CO_SUM perform **collective** operations. These are for coarray parallelism: they compute values across all images in the current team, without explicit synchronisation.

  All of these subroutines have optional STAT and ERRMSG arguments. On successful execution, the STAT argument is assigned the value zero and the ERRMSG argument is left unchanged. If an error occurs, a positive value is assigned to STAT and an explanatory message is assigned to ERRMSG. Only the errors STAT_FAILED_IMAGE and STAT_STOPPED_IMAGE are likely to be able to be caught in this way. Because there is not full synchronisation (see below), different images may receive different errors, or none at all. If an error occurs and STAT is not present, execution is terminated. Note that if the actual arguments for STAT or ERRMSG are optional dummy arguments, they must be present on all images or absent on all images.

  A reference (CALL) to one of these subroutines is **not** an image control statement, does not end the current segment, and does not imply synchronisation (though some partial synchronisation will occur during the computation). However, such calls are only permitted where an image control statement is permitted.

  Each image in a team must execute the same sequence of CALL statements to collective subroutines as the other images in the team. There must be no synchronisation between the images at the time of the call; the invocations must come from unordered segments.

  All collective subroutines have the first argument "A", which is INTENT(INOUT), and must not be a coindexed object. This argument contains the data for the calculation, and must have the same type, type parameters, and shape on all images in the current team. If it is a coarray that is a dummy argument, it must have the same ultimate argument on all images.

  SUBROUTINE CO_BROADCAST (A, SOURCE_IMAGE *[*, STAT, ERRMSG *]*)

  A : variable of any type; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

  SOURCE_IMAGE : integer scalar, in the range one to NUM_IMAGES(), this argument must have the same value on all images in the current team;

  STAT (optional) : integer scalar variable, not coindexed;

ERRMSG (optional) : character scalar variable of default kind, not coindexed.

The value of argument A on image SOURCE_IMAGE is assigned to the argument A on all the other images.

SUBROUTINE CO_MAX (A [, RESULT_IMAGE, STAT, ERRMSG])

A: variable of type Integer, Real, or Character; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

RESULT_IMAGE (optional) : integer scalar, in the range one to NUM_IMAGES(), this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

STAT (optional) : integer scalar variable, not coindexed;

ERRMSG (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes the maximum value of A across all images; if A is an array, the value is computed elementally. If RESULT_IMAGE is present, the result is assigned to argument A on that image, otherwise it is assigned to argument A on all images.

SUBROUTINE CO_MIN (A [, RESULT_IMAGE, STAT, ERRMSG])

A: variable of type Integer, Real, or Character; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

RESULT_IMAGE (optional) : integer scalar, in the range one to NUM_IMAGES(), this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

STAT (optional) : integer scalar variable, not coindexed;

ERRMSG (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes the minimum value of A across all images; if A is an array, the value is computed elementally. If RESULT_IMAGE is present, the result is assigned to argument A on that image, otherwise it is assigned to argument A on all images.

SUBROUTINE CO_REDUCE (A, OPERATION [, RESULT_IMAGE, STAT, ERRMSG])

A: non-polymorphic variable of any type; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

OPERATION : pure function with exactly two arguments; the dummy arguments of OPERATION must be non-allocatable, non-optional, non-pointer, non-polymorphic dummy variables, and each argument and the result of the function must be scalar with the same type and type parameters as A;

RESULT_IMAGE (optional) : integer scalar, in the range one to NUM_IMAGES(), this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

STAT (optional) : integer scalar variable, not coindexed;

ERRMSG (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes an arbitrary reduction of A across all images; if A is an array, the value is computed elementally. The reduction is computed starting with the set of corresponding values of A on all images; this is an iterative process, taking two values from the set and converting them to a single value by applying the OPERATION function; the process continues until the set contains only a single value — that value is the result. If RESULT_IMAGE is present, the result is assigned to argument A on that image, otherwise it is assigned to argument A on all images.

`SUBROUTINE CO_SUM ( A [, RESULT_IMAGE, STAT, ERRMSG ] )`

`A`: variable of type Integer, Real, or Complex; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if `A` is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

`RESULT_IMAGE` (optional) : integer scalar, in the range one to `NUM_IMAGES()`, this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

`STAT` (optional) : integer scalar variable, not coindexed;

`ERRMSG` (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes the sum of `A` across all images; if `A` is an array, the value is computed elementally. If `RESULT_IMAGE` is present, the result is assigned to argument `A` on that image, otherwise it is assigned to argument `A` on all images.

12 References

The Fortran 2018 standard, ISO/IEC 1539-1:2018(E), is available from ISO as well as from many national standards bodies. A number of books describing the new standard are available; the recommended reference book is "Modern Fortran Explained (Incorporating Fortran 2018)" by Metcalf, Reid & Cohen, Oxford University Press, 2018 (ISBN 978-0-19-881188-6).