Modern Fortran: Features for High-Performance Computing

Steve Lionel, Convenor, ISO/IEC JTC1/SC22/WG5 Fortran Standards Committee
https://stevelionel.com/drfortran

May 2021

Agenda

- Fortran through the ages
- A brief introduction to Fortran
- Object-oriented features
- C Interoperability
- Parallelization (DO Concurrent Coarrays)
- Future standardization plans

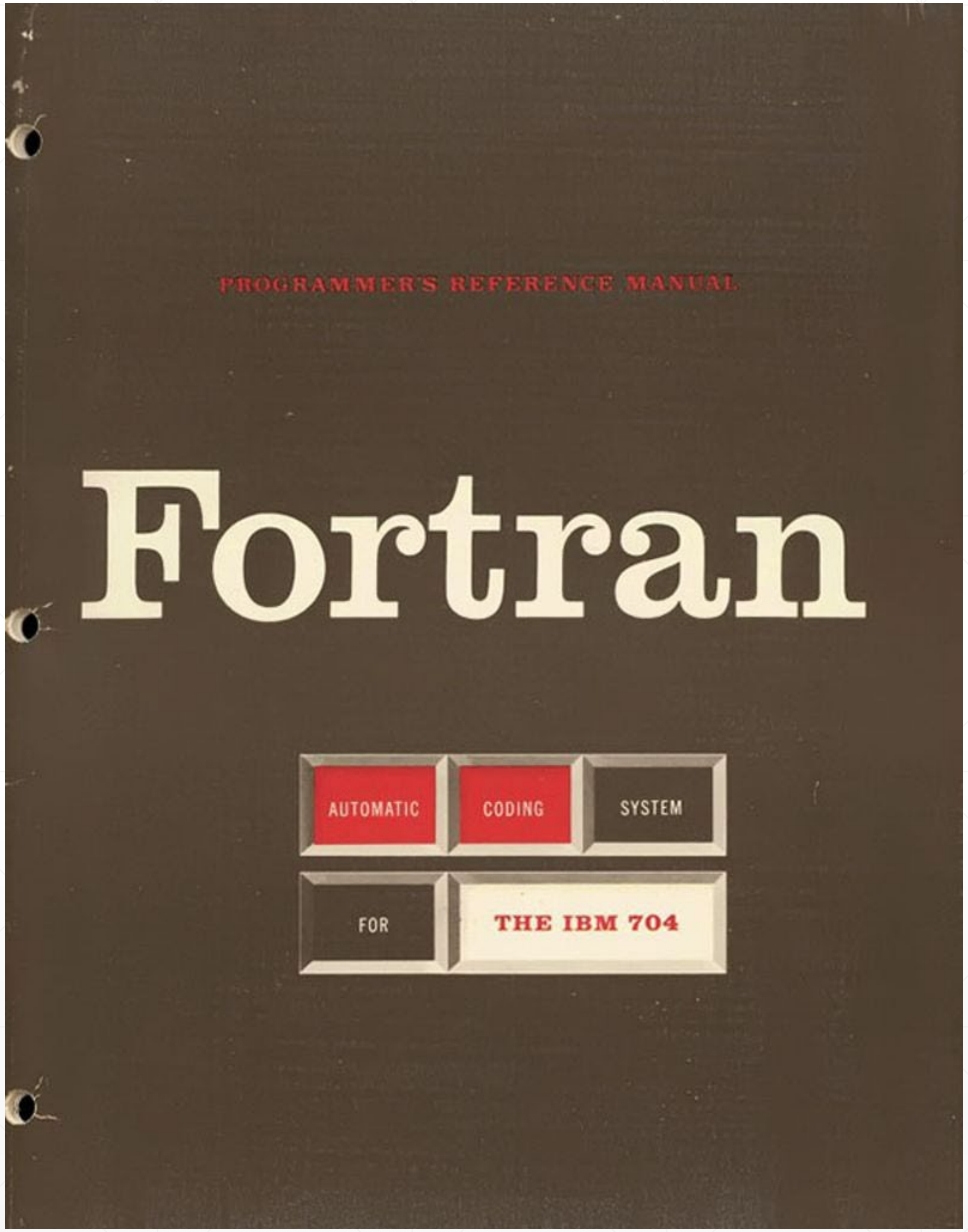
During the presentation…

- If you have questions, please use the Zoom Chat window. I will pause at times and look to see questions that were asked
- We will have a break about half-way through
- There will be time at the end for general questions
- Feel free to email me afterward with questions about any of these topics (steve@stevelionel.com – also @DoctorFortran on Twitter.
- “DF:” indicates related post on my blog, https://stevelionel.com/drfortran

Fortran through the ages

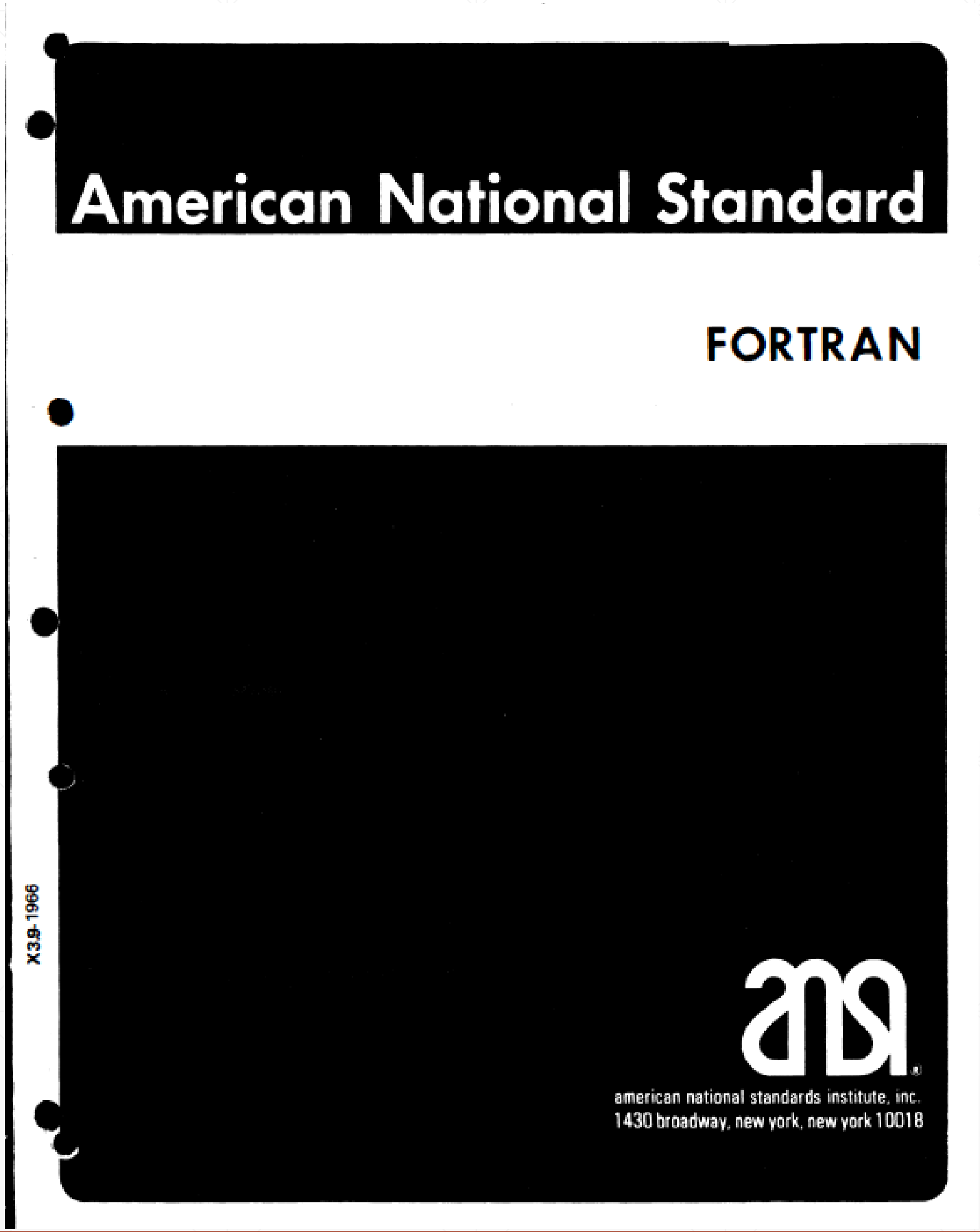
History of Fortran

- 1954 - Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN.
- 1956 - The FORTRAN Automatic Coding System for the IBM 704



ANSI FORTRAN 66

Published March 1966


American National Standard

FORTRAN

X3.9-1966

american national standards institute, inc.
1430 broadway, new york, new york 10018

Later Standards

- FORTRAN 77 (1978)
- Fortran 90 (1991)
- Fortran 95 (1997)
- Fortran 2003 (2004)
- Fortran 2008 (2010)
- Fortran 2018 (2018)

The Fortran Standard Through the Years

How is a new Fortran standard made?

- International Fortran committee is ISO/IEC JTC1/SC22/WG5
- Experts from individual countries make up National Bodies
- WG5 determines general content of the standard
- Development of features is done by the US National Body (INCITS PL22.3, informally “J3”)
- All WG5 members vote to approve a “Final Committee Draft”
- Fortran 2018 was published November 2018
- Next revision working title “Fortran 202X”

What the standard does and doesn't say

- The standard describes how a standard-conforming program is interpreted
- Compilers can (and do) assign meanings to syntax and relationships not specified in the standard
- Compilers need only have the capability of reporting code that does not meet numbered syntax rules or constraints
- All compilers have extensions – make sure you understand what you are using
- Obsolescent and deleted features - avoid if possible

A Brief Introduction to Fortran

Why Fortran

- Modern language features
- Compatibility with large body of existing code
- Robust array operations
- Standard gives compilers flexibility to optimize
- Can mix Fortran with other languages (C interoperability)
- Built-in parallel features
- Choice of implementations

Free-form Source

- Free source form introduced in Fortran 90
  - Source lines may be up to 132 characters long
  - Statement can start in any column
  - Continuation indicated by & at the end of a line to be continued – next line may start with & - required in some cases
  - Up to 255 continuation lines
  - Semicolon (;) ends a statement – can have multiple statements on a line separated by ;
  - Blanks are significant and must appear between adjacent keywords, with some exceptions allowed (END IF or ENDIF, DOUBLE PRECISION or DOUBLEPRECISION, etc.)
  - ! Indicates rest of line is a comment
  - File type .f90 is best to use for free source form (DF: “Source Form Just Wants to be Free”)
  - Use free source form for all new code!

Fixed-form Source

- Obsolete – compilers will complain if standards check enabled
- 72-character lines
- Numeric label field in columns 1-5
- Continuation indicator nonblank in column 6
- Statement in columns 7-72
- Blanks NOT significant (except in character constants)
- Semicolon can be used to end/separate statement as in free-form
- .f or .for file types typical
- DO NOT USE IN NEW CODE!

Program Units

- PROGRAM defines the main program
- SUBROUTINE or FUNCTION
- Internal subroutine or function declared after CONTAINS
- MODULEs and SUBMODULEs (DF: We All Live in a Yellow Submodule)
- BLOCK DATA (obsolete)

Identifiers

- Alphanumeric (Letters (A-Z), Digits (0-9), underscore)
  - $ is not standard in identifiers, but is a common extension
- Must start with a letter
- Up to 63 characters long
- NOT case-sensitive
  - VARNAME, varname, VarName all the same
  - Keywords are also not case-sensitive
- No reserved words (DF: No Reserve)

Types

- Intrinsic types INTEGER, REAL, COMPLEX, LOGICAL, CHARACTER
- KIND numbers distinguish variants of an intrinsic type. Example: INTEGER(4) or INTEGER(KIND=4)
  - KIND numbers are implementation-dependent, not always tied to storage size
  - COMPLEX kinds are the same as the component REAL kinds
- SELECTED_INT_KIND, SELECTED_REAL_KIND, SELECTED_CHAR_KIND intrinsic functions to specify kind that meets precision, range and/or character set requirements
- CHARACTER types have fixed lengths (except deferred-length allocatable)
- DF: It Takes All KINDs

Derived Types

- User-defined types
- `TYPE mytype;` *component-list*`; END TYPE mytype`
- Components can be intrinsic or derived types
- % separates components in a reference (A%B%C)
- No unions

Constants and Operators

- Intrinsic type constants can have optional kind (3.1415926535897_8, 42_4)
  - Kind specifier can be named constant (3.1416026535897_DP)
- Array constructors: [1,2,3,4]
- Structure constructors: mytype(3,’ABC’,.TRUE.)
- Exponentiation is **
- Comparison: <, >, <=, >=, ==, /= (older: .LT., .GT., .LE., .GE., .EQ., .NE.)
- Logical: .AND., .OR., .NOT., .EQV., .NEQV. (DF: It’s Only LOGICAL)
- User-defined: Example: .DOT_PRODUCT.
- DF: Order! Order!

Arrays

- Array types
  - Explicit-shape – `REAL :: X(10)`
  - Adjustable – `REAL :: X(J)`, where `J` is a dummy argument/COMMON/MODULE variable
  - Assumed-size – `REAL :: X(*)`
  - Assumed-shape – `REAL :: X(:)`, where `X` is dummy argument
  - Deferred-shape – `REAL :: X(:)`, where `X` is allocatable or pointer
  - Implied-shape – `REAL, PARAMETER :: X(*) = [1,2,3]`
  - Assumed-rank – `REAL :: X(..)`, where `X` is dummy argument
- Maximum of 15 dimensions
- Default lower bound is 1, can be changed (`REAL :: X(0:8)`)

Arrays

- Whole-array assignment - A = B
- Array operations – A = B + C
- Array sections - A(1:100:2)
- Vector subscripts – A([2,5,4,7])
- Intrinsics operate on arrays - (MAXLOC, IALL, MATMUL, DOT_PRODUCT, many more)
- Elemental functions operate on scalars or arrays (or write your own!)

Control Flow

- DO
  - Counted DO: `label: DO I=1,10 .. END DO`
  - Tested DO: `DO WHILE (T > 0) .. END DO`
  - Infinite DO: `DO .. END DO`
  - Parallel allowed DO: `DO CONCURRENT ((I=1:10, J=1:10, A(I) > 0.0 .AND. B(J) < 1.0)`
  - Skip to next iteration: `CYCLE [label]`
  - Leave this loop (or construct): `EXIT [label]`
- Obsolete and Deleted forms
  - `DO 10 I=1,10 .. 10 CONTINUE`
  - `DO 10 I=1,10 .. 10 J = J + I`
  - DF: Hey! Who are you calling Obsolescent?

Control Flow

- `IF` (*expression*) *statement*

- `IF` (*expression*)
  `THEN`
  statements
  `[ELSE`
  statements]
  `END IF`

- SELECT CASE (N)

```
SELECT CASE (N)
  CASE (:-1)
    SIGNUM = -1
  CASE (0)
    SIGNUM = 0
  CASE (1:)
    SIGNUM = 1
  CASE DEFAULT
    ERROR STOP
END SELECT
```

Modules

- Separately compiled collection of declarations and/or procedures
- Example:

```
module mymod
integer, parameter :: widget = 23
type things
  integer :: thing_id
end type things
contains
function get_thing (thing_code)
…
end function get_thing
end module mymod
```

Submodules

- Separate interface from implementation
- Reduce or eliminate “compilation cascade”
- DF: We All Live in a Yellow Submodule

Explicit Interface

- Before Fortran 90, all procedure interfaces were implicit
- Explicit interface declares characteristics of procedure
- Explicit interface automatic for module procedures and internal procedures
- INTERFACE block can be used to create an explicit interface
- Some types of procedures require an explicit interface to be visible
- IMPLICIT NONE (EXTERNAL) forces you to explicitly declare all procedures
- DF: Doctor Fortran Gets Explicit (and Doctor Fortran Gets Explicit – Again!)

Pointer and Allocatable

- POINTER
  - Contains shape and dynamic type information
  - Reference to pointer is to its data, except in pointer assignment (and when passing a pointer to another pointer)
  - Can pointer-assign to anything with the TARGET attribute
  - Can be discontiguous array (stride other than 1)
  - No garbage collection – programmer responsible for avoiding leaks
  - Assumption that pointer objects can be aliased

Pointer and Allocatable

- ALLOCATABLE
  - Contains shape and dynamic type information
  - Always contiguous
  - Never aliased
  - Compiler ensures no leaks
  - Intrinsic assignment automatically reallocates if needed
  - Deferred-length allocatable character serves as varying length strings
    - ```
      CHARACTER(:), ALLOCATABLE :: STR
      STR = 'ABCDE' ! STR is length 5
      STR = 'FGHIJKL' ! STR is now length 7
      ```
      ```

Input/Output

- ACCESS
  - SEQUENTIAL – sequence of records that can be variable length
  - DIRECT – fixed-length records randomly accessed by record number
  - STREAM – C-like sequence of bytes, can be repositioned
- FORM
  - FORMATTED – text representation
  - UNFORMATTED – binary representation
- Internal I/O
  - READ from, WRITE to CHARACTER variables

Object-Oriented Features

Type Extension

- Create a new type by extending an existing derived type
- Most derived types can be extended
- Type that is extended is the **parent type**
- Extended type inherits all its parent's components
- The parent type is itself a component of the extended type
- DF: Not My TYPE

Type Extension Example

```
type :: parent
  integer :: p1
end type parent
type, extends(parent) :: child
  real :: c2
end type child
type(child) :: kid
```

Available components: `kid%c2, kid%p1, kid%parent, kid%parent%p1`

Polymorphism

- Polymorphic variables have a declared type and a dynamic type
- `CLASS(parent), POINTER :: p` can point to an object of type `parent` or any of its extensions
- Allocatable polymorphic variables can have their dynamic type specified in the ALLOCATE statement: `ALLOCATE (child::p)`
- A polymorphic variable is type-compatible with an object of the same **declared type** or any of its extensions
- `CLASS(*)` means **unlimited polymorphic** – everything is type-compatible with it, but it has no declared type
- `SELECT TYPE` construct to choose based on dynamic type

Type-bound Procedures

```
MODULE MYMOD
TYPE POINT
REAL :: X, Y
CONTAINS
PROCEDURE, PASS :: RADIUS => POINT_RADIUS
END TYPE POINT

CONTAINS

FUNCTION POINT_RADIUS (THIS)
REAL :: POINT_RADIUS
CLASS(POINT), INTENT(IN) :: THIS
POINT_RADIUS = SQRT((THIS%X)**2 + (THIS%Y)**2)
END FUNCTION POINT_RADIUS
END MODULE MYMOD

PROGRAM TEST
USE MYMOD
TYPE(POINT) :: P
P = POINT (2.0,3.0)
PRINT *, P%RADIUS()
END PROGRAM TEST
```

C Interoperability

C Interoperability

- Standard features to call C (or C-like languages) from Fortran, and to call Fortran from C
- Standard features to share data between Fortran and C
- Definitions and restrictions to ensure common interpretation
- Standard talks about "companion C processor"
- Handles naming, data types and layout, argument passing
- DF: I Can C Clearly Now

Interoperable Types

- Intrinsic types of kinds with supported equivalents in C
- Intrinsic module ISO_C_BINDING defines named constants for various C kinds (C_INT, C_FLOAT, C_SIZE_T, C_CHAR, C_LONG_LONG, etc.)
- If there is no equivalent kind, value of constant is -1
- Derived types with BIND(C) attribute are interoperable
  - All components must be interoperable
  - No pointer, allocatable or coarray components
  - Layout in memory matches that of companion C processor

Interoperable Procedures

- Procedure declared with BIND(C) attribute in explicit interface
- Optional NAME= specifies case-sensitive name
- Name "decoration" done as the C processor would
- All dummy arguments must be interoperable
- Fortran procedures can also be interoperable
- C strings interoperable with array of single characters

Fortran 2018 Enhancements

- In Fortran 2003 and 2008, these kinds of dummy arguments were not interoperable:
  - Assumed-shape arrays
  - Assumed-size arrays
  - Character length other than 1
  - Allocatable or pointer variables
- In Fortran 2018, all of these are now interoperable when a “C Descriptor” is passed

C Descriptor

- A C descriptor includes:
  - Attribute code (POINTER, ALLOCATABLE, OTHER)
  - Data type
  - Base address
  - Element length
  - Rank
  - Bounds and extents

C Descriptors

- Fortran creates and passes C descriptors to routines declared as BIND(C)
- C code can operate on C descriptors with CFI_xxx functions
- C code can create C descriptors and pass to Fortran
  - Fortran procedure must have `BIND(C)` attribute
- Descriptor layout, constants, functions declared in `ISO_Fortran_binding.h` supplied with each Fortran compiler
  - C descriptors not interoperable with other Fortran compilers
- Be careful about `CHARACTER(*)` dummy arguments – passed by C descriptor!

```c
#include "ISO_Fortran_binding.h"
#include <memory.h>
#include <stdio.h>

extern "C" void greetings(CFI_cdesc_t * descr);

int main()
{
        int status;
        CFI_CDESC_T(0) cdesc;

        // Create our own local descriptor for an allocatable string
        status = CFI_establish((CFI_cdesc_t *)&cdesc, NULL,
                               CFI_attribute_allocatable,
                               CFI_type_char, 1, 0, NULL);
        //Allocate the string to length 7
        status = CFI_allocate((CFI_cdesc_t *)&cdesc, NULL, NULL, 7);
        // Copy in 'Hello, '
        memcpy(cdesc.base_addr, "Hello, ", 7);
        // Call Fortran to append to the string and print it
        greetings((CFI_cdesc_t *)&cdesc);
        printf("Length is now %zd\n", cdesc.elem_len);
        status = CFI_deallocate((CFI_cdesc_t *)&cdesc);
}
```

```
subroutine greetings (string) bind(C)
    implicit none
    character(:), allocatable :: string

    string = string // 'World!'
    print *, string
    print *, ‘Length is now’, len(string)
    end subroutine greetings
```

```
Hello, World!
Length is now 13
```

Assumed Type

- Syntax is `TYPE(*)`
- Unlimited polymorphic - has no declared type
- May be used only for dummy arguments
- Like C `void`
- Limited use in Fortran code

Allocatable Dummy Arguments

- `ALLOCATABLE, INTENT(OUT)` dummy arguments get deallocated on entry to a Fortran procedure
- In Fortran 2018, a `BIND(C)` procedure can now have such an argument
- Fortran processor is required to do the deallocation on the call

More F2018 Interoperability Changes

- A Fortran procedure with a `CONTIGUOUS` dummy argument must be able to handle a C descriptor for a non-contiguous array
- Interoperable procedures may now have `OPTIONAL` dummy arguments
- `ASYNCHRONOUS` attribute extended to data access other than input/output

Assumed Rank

- Syntax is `DIMENSION(..)`
- May be used only for dummy arguments
- New `SELECT RANK` construct for use in Fortran code
  - `RANK(n)`
  - `RANK(*)` for assumed-size array
  - `RANK DEFAULT`

DO CONCURRENT

DO CONCURRENT

- Replaces FORALL from Fortran 95
- Allows for parallelization, does not require it
- `DO CONCURRENT` *concurrent-header concurrent-locality*
  block
  `END DO`
- *concurrent-header*: `([`*type-spec*`::]` *control-list* `[,` *scalar-mask-expr*`])`
- *control-list*: *index-name* = *limit* `:` *limit* [`:` *step*]
- Example: `DO CONCURRENT (I=1:10, J=1:10, A(I) > 0.0 .AND. B(J) < 1.0)`

DO CONCURRENT Locality Specifications

- Tells the compiler which variables are local to each iteration
- If a variable is not named, compiler can try to figure it out
- Choices:
  - `LOCAL` (*variable-name-list*)
  - `LOCAL_INIT` (*variable-name-list*)
  - `SHARED` (*variable-name-list*)
  - `DEFAULT (NONE)`

DO CONCURRENT Locality Example

```
real :: a(:), b(:), x
…
do concurrent (i=1:size(a)) local (x) shared (a,b)
  if (a(i) > 0) then
    x = sqrt(a(i))
    a(i) = a(i) - x**2
  end if
  b(i) = b(i) - a(i)
end do
…
```

Example from Modern Fortran Explained, 8th Edition

Coarrays

Summary of coarray model

- SPMD - Single Program, Multiple Data
- Replicated to a number of **images** (probably as executables)
- Number of images fixed during execution
- Each image has its own set of variables
- Coarrays are like ordinary variables but have second set of subscripts [ ] for access between images
- Images mostly execute asynchronously
- Synchronization: `sync all`, `sync images`, `lock`, `unlock`, `critical construct`, `allocate`, `deallocate`
- Intrinsics: `this_image`, `num_images`, `image_index`

Examples of coarray syntax

```
real,save :: r[*], s[0:*] ! Scalar coarrays
real,save :: x(n)[*]      ! Array coarray
type(u),save :: u2(m,n)[np,*]
! Coarrays always have assumed cosize
! (equal to number of images)
real :: t                   ! Local variable
integer p, q, index(n)    ! Local variables
           :
t = s[p]
x(:) = x(:)[p]
! Reference without [] is to local object
x(:)[p] = x(:)
u2(i,j)%b(:) = u2(i,j)[p,q]%b(:)
```

Implementation model

- Usually, each image resides on one core.
- However, several images may share a core (e.g. for debugging) and one image may execute on a node (e.g. with OpenMP).
- A coarray has the same set of bounds on all images, so the compiler may arrange that it occupies the same set of addresses within each image (known as ***symmetric memory***).
- This allows each image to calculate the memory address of an element on another image.

Synchronization

- The images execute asynchronously. If syncs are needed, the user supplies them explicitly.
- Barrier on all images
  - `sync all`
- Wait for others
  - `sync images` (*image-set*)
- Limit execution to one image at a time
  - ```
    critical
       block
    end critical
    ```
- These are known as **image control** statements
    ```

Execution segments

- On an image, the statements executed up to the first image control statement or after one and up to the next is known as a **segment**.
- For example, this code reads a value on image 1 and broadcasts it.

```
  :                           ! Segment 1
sync all                      ! Segment 1
if(this_image()==1)then       ! Segment 2
   read (*,*) p               !    :
   do i = 2, num_images()     !    :
      p[i] = p                !    :
   end do                     !    :
end if                        !    :
sync all                      ! Segment 2
  :                           ! Segment 3
```

Execution segments (cont)

- The normal rules of statement execution on a single image and the synchronization statements together ensure a partial ordering of all the segments.
- **Important rule:** if a variable is defined in a segment, it must not be referenced, defined, or become undefined in a another segment unless the segments are ordered.
- It is up to the programmer to ensure this.

Dynamic coarrays

- Only dynamic form: the allocatable coarray.
- ```
  real, allocatable :: a(:)[:], s[:,:]
      :
  allocate ( a(n)[*], s[-1:p,0:*] )
  ```
- The bounds, cobounds, and length parameters must not vary between images.
- All images synchronize at an `allocate` or `deallocate` statement so that they can all perform their allocations and deallocations in the same order (for symmetric memory).
  ```

Coarray dummy arguments

- A dummy argument may be a coarray. It may be of explicit shape, assumed size, assumed shape, or allocatable.
- ```
  subroutine subr(n,w,x,y,z)
     integer :: n
     real :: w(n)[n,*]  ! Explicit shape
     real :: x(n,*)[*]  ! Assumed size
     real :: y(:,:)[*]  ! Assumed shape
     real, allocatable :: z(:)[:,:]
  ```
- There are rules to ensure that copy-in copy-out of a coarray is never needed.
  ```

Structure components

- A coarray may be of a derived type with allocatable or pointer components.
- Provides a simple but powerful mechanism for cases where the size varies from image to image, avoiding loss of optimization.
- Pointers must have targets in their own image:
- `q => z[i]%p         ! Not allowed`
- `allocate(z[i]%p) ! Not allowed`

Teams

In Fortran 2008, program images were uniformly numbered starting at 1

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32

Teams

Teams allow splitting images into groups

Teams

- Needed for independent computations on subsets of images.
- Code that has been written and tested on whole machine should run on a team.
- Therefore, image indices need to be relative to the team.
- Collective activities, including syncs and allocations, need to be relative to the team.

`team_type` and `form team`

- The intrinsic module `iso_fortran_env` contains a derived type `team_type`. A scalar object of this type identifies a team of images.
- The same `form team` statement must be executed on all images of a team to form subteams
- ```
        form team(number,new_team)
  ```
- Images with the same value of `number` form a new team.
- All images of the current team synchronize.
  ```

`change team` construct

```
change team (team,local[*]=>coarray)
   ! Block executed as a team
   if(team_number()==1) then ! New intrinsic
         : ! Code for team 1
   else
         :
end team
```

- Associating `local` with `coarray` allows corank and cobounds to change. Other attributes are unchanged.
- The new teams synchronize at `change team` and `end team`.
- Changing teams is likely to be costly – avoid doing it often.

Accessing another team

```
real, save :: a(n)[*]
type(team_type) :: initial, block
initial = get_team() ! New intrinsic
i = ...
form team(i,block)
change team (block)
   :
   sync team(initial) ! New statement
   a(k) = a(1)[me+1,team=initial]
end team
```

Collectives

- The collective subroutines are:
- `co_broadcast, co_max, co_min, co_sum, co_reduce.`
- Invoked by the same statement on all images of the team and involve synchronization within them, but not necessarily at start and end.
- The main argument is not required to be a coarray.

CO_REDUCE Example

```
SUBROUTINE co_all(boolean)
LOGICAL, INTENT(INOUT) :: Boolean
CALL CO_REDUCE(boolean,both)

CONTAINS

PURE FUNCTION both(lhs,rhs) RESULT(lhs_and_rhs)
LOGICAL, INTENT(IN) :: lhs,rhs
LOGICAL :: lhs_and_rhs
lhs_and_rhs = lhs .AND. rhs
END FUNCTION both

END SUBROUTINE co_all
```

Events

- Events are useful if one or more images need to do something before another image can continue.
- For example, in the multifrontal method for factorizing a sparse matrix, work at a node of the assembly tree has to wait for all the work at its child nodes to be completed.

Events

- An event variable is a scalar coarray of type `event_type`. It contains a count which increases by one each time the event is "posted".

```
use iso_fortran_env
type(event_type), save :: event[*]
  :
event post(event[i]) ! Atomic
  :
if(this_image()==i) then
  event wait(event)
  ! Waits until count >= 1, then atomically
  ! decreases it by 1 and continues
```

Events Example (outline)

```
PROGRAM TREE
USE, INTRINSIC :: ISO_FORTRAN_ENV
INTEGER, ALLOCATABLE :: NODE (:) ! Tree nodes that this image handles.
INTEGER, ALLOCATABLE :: NC (:) ! NODE(I) has NC(I) children.
INTEGER, ALLOCATABLE :: PARENT (:), SUB (:)
  ! The parent of NODE (I) is NODE (SUB (I)) [PARENT (I)].
TYPE (EVENT_TYPE), ALLOCATABLE :: DONE (:) [:]
INTEGER :: I, J, STATUS
  ! Set up the tree, including allocation of all arrays.
DO I = 1, SIZE (NODE)
  ! Wait for children to complete
  IF (NC (I) > 0) THEN
    EVENT WAIT (DONE (I), UNTIL_COUNT=NC (I), STAT=STATUS)
    IF (STATUS/=0) EXIT
  END IF

  ! Process node, using data from children.
  IF (PARENT (I)>0) THEN
    ! Node is not the root.
    ! Place result on image PARENT (I) for node NODE (SUB) [PARENT (I)]
    ! Tell PARENT (I) that this has been done.
    EVENT POST (DONE (SUB (I)) [PARENT (I)], STAT=STATUS)
    IF (STATUS/=0) EXIT
  END IF
END DO
END PROGRAM TREE
```

`failed_images` intrinsic function

- `failed_images()`
  - Returns an integer array holding image indices of known failed images in the current team.
- `failed_images(team)`
  - Returns an integer array holding image indices of known failed images in `team`.

Testing for failed images in image control statements

```
parent = get_team()
change team (team_a)
       :
   sync_all(parent,stat=st)
   if (st==stat_failed_image) exit
end team
sync all(stat=st)
if (st==stat_failed_image) then
   : Deal with failure
end if
```

Testing for failed image in a remote reference

```
use iso_fortran_env
   :
a = b[image,stat=st]
if (st==stat_failed_image) then
   : Deal with failure
end if
```

Advantages of coarrays

- References to local data are obvious as such.
- Easy to maintain code - more concise than MPI and easy to see what is happening
- Integrated with Fortran - type checking, type conversion on assignment, ...
- The compiler can optimize communication
- Local optimizations still available
- Does not make severe demands on the compiler, e.g. for coherency.

Coarray Example

```
! This program demonstrates using Fortran coarrays to implement the classic
! method of computing the mathematical value pi using a Monte Carlo technique.
! A good explanation of this method can be found at:
! http://www.mathcs.emory.edu/~cheung/Courses/170/Syllabus/07/compute-pi.html
program mcpi
implicit none

! Declare kind values for large integers, single and double precision
integer, parameter :: K_BIGINT = selected_int_kind(15)
integer, parameter :: K_DOUBLE = selected_real_kind(15,300)

! Number of trials per image. The bigger this is, the better the result
! This value must be evenly divisible by the number of images.
integer(K_BIGINT), parameter :: num_trials = 1200000000_K_BIGINT

! Actual value of PI to 18 digits for comparison
real(K_DOUBLE), parameter :: actual_pi = 3.141592653589793238_K_DOUBLE

! Declare scalar coarray that will exist on each image
integer(K_BIGINT) :: total[*] ! Per-image subtotal

! Local variables
real(K_DOUBLE) :: x,y
real(K_DOUBLE) :: computed_pi
integer :: I
integer(K_BIGINT) :: bigi
integer(K_BIGINT) :: clock_start,clock_end,clock_rate
```

Coarray Example (page 2)

```
! Image 1 initialization
if (THIS_IMAGE() == 1) then
    ! Make sure that num_trials is divisible by the number of images
    if (MOD(num_trials,INT(NUM_IMAGES(),K_BIGINT)) /= 0_K_BIGINT) &
        error stop "Number of trials not evenly divisible by number of images!"
    print '(A,I0,A,I0,A)', "Computing pi using ",num_trials, &
        " trials across ",NUM_IMAGES()," images"
    call SYSTEM_CLOCK(clock_start)
end if

! Set the initial random number seed to an unpredictable value, with a different
! sequence on each image.
call RANDOM_INIT (REPEATABLE=.FALSE.,IMAGE_DISTINCT=.TRUE.)

! Initialize our subtotal
total = 0_K_BIGINT

! Run the trials, with each image doing its share of the trials.
!
! Get a random X and Y and see if the position
! is within a circle of radius 1. If it is, add one to the subtotal
do bigi=1_K_BIGINT,num_trials/int(NUM_IMAGES(),K_BIGINT)
    call RANDOM_NUMBER(x); call RANDOM_NUMBER(y)
    if ((x*x)+(y*y) <= 1.0_K_DOUBLE) total = total + 1_K_BIGINT
end do

print *, "Image ", this_image(), " found ", total, " values"
```

Coarray Example (page 3)

```
! Wait for everyone
sync all

! Image 1 end processing
if (this_image() == 1) then
    ! Sum all of the images' subtotals
    do i=2,num_images()
        total = total + total[i]
    end do

    ! total/num_trials is an approximation of pi/4
    computed_pi = 4.0_K_DOUBLE*(REAL(total,K_DOUBLE)/REAL(num_trials,K_DOUBLE))
    print '(A,G0.8,A,G0.3)', "Computed value of pi is ", computed_pi, &
        ", Relative Error: ",ABS((computed_pi-actual_pi)/actual_pi)

    ! Show elapsed time
    call SYSTEM_CLOCK(clock_end,clock_rate)
    print '(A,G0.3,A)', "Elapsed time is ", &
        REAL(clock_end-clock_start)/REAL(clock_rate)," seconds“
end if

end program mcpi
```

Running the coarray example

```
Computing pi using 1200000000 trials across 12 images
 Image  8  found  78545883  values
 Image  11  found  78539293  values
 Image  7  found  78538166  values
 Image  6  found  78533956  values
 Image  10  found  78541985  values
 Image  12  found  78551690  values
 Image  5  found  78536524  values
 Image  9  found  78538020  values
 Image  3  found  78541247  values
 Image  4  found  78539400  values
 Image  2  found  78535164  values
 Image  1  found  78534353  values

Computed value of pi is 3.1415856, Relative Error: .224E-05

Elapsed time is 4.31 seconds
```

Fortran 202X

Future Revisions

- Next revision is informally called Fortran 202X
- Goal is to have it published no later than 2023
- Six-month survey of users 2017-2018
  - Results in WG5 document N2147
- After that, Fortran 202Y

Fortran 202X Features

(nn-nnn refers to papers at https://j3-fortran.org/)

- Add optional argument to C_F_POINTER to specify lower bounds (19-238r1)
- Longer source lines and statement length (19-138r1)
  - Require reporting of ignored characters after line length limit, if any (19-149r1)
- Trigonometric functions in degrees (SIND, COSD, etc.) (19-203r1)
- Trigonometric functions scaled by π (SINPI, COSPI, etc.) (19-204r1)
- SELECTED_LOGICAL_KIND intrinsic (19-147r1)
- LOGICALnn constants in ISO_FORTRAN_ENV (19-139r1)

Fortran 202X Features Continued

- SPLIT function splits strings into tokens based on separators (19-254r1)
- C_F_STRPOINTER and F_C_STRING for help with C strings (19-197r3)
- AT format specifier for trimming strings (19-137r2)
- Format control over leading zeros for reals (19-156r1)
- Allow arrays of derived type with coarray components (19-250r1)
- Put with notify for coarrays (19-259r1)
- Automatically allocate deferred-length character in internal WRITE and IOMSG/ERRMSG (19-252r2)

Fortran 202X Features Continued

- Reduction specifier in DO CONCURRENT (19-255r2)
- Allow BOZ constants in more places (19-256r2)
- SIMPLE procedures are PURE with more restrictions (19-201r1)
- TYPEOF, CLASSOF intrinsics to help with generic programming (19-142r1)
- Rank-agnostic allocation and pointer assignment (20-120r1)
- BOUNDS() and RANK() specifiers for DIMENSION attribute (19-202r2)
- Rank-agnostic array notation (20-144r2)

Approved F202X Features not yet finished

- Protected components (20-106)
- Typed enumerators (19-249r1)
- Conditional expressions
- Short-circuit logical operators (18-239)
- Variant of INTENT that applies to a pointer target (18-144r1)

More Information

- WG5 web site https://wg5-fortran.org
  - Documents > N2161 The New Features of Fortran 2018
  - Fortran Standards > Fortran 2018
- J3 (PL22.3) web site https://j3-fortran.org
  - Repository for papers related to technical content of the standard
  - 18-007r1 is the committee reference for Fortran 2018
- Doctor Fortran blog https://stevelionel.com/drfortran
- Ideas for future revisions https://github.com/j3-fortran/fortran_proposals
- Fortran Discourse https://fortran-lang.discourse.group/

Questions?

- Use Raise Hand feature or ask in the Chat window