



Pekka Manninen
Sami Saarinen
David Henty



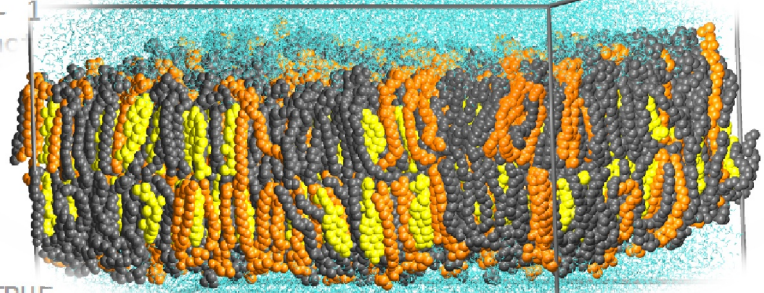
Fortran 2003/2008

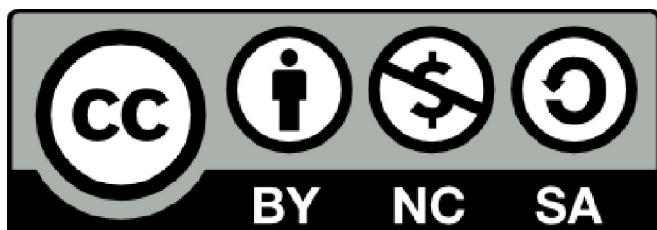
September 11-13, 2012

PRACE Advanced Training Centre

CSC – IT Center for Science Ltd, Finland

```
DO imolecule_kind_a=1,nmolecule_kind
  molecule_kind_a => molecule_kind_set(imolecule_kind_a)
  CALL get_molecule_kind(molecule_kind=molecule_kind_a, molecule_list=molecule_list_a,&
    natom=natom_mol_a)
DO imolecule_a=1,SIZE(molecule_list_a)
  imol_a = imol_a + 1
  DO iatom_mol_a=1,natom_mol_a
    iatom_a = molecule_set(molecule_list_a(imolecule_a))%first_atom + &
      iatom_mol_a - 1
    imol_b = 0
    DO imolecule_kind_b=1,nmolecule_kind
      molecule_kind_b => molecule_kind_set(imolecule_kind_b)
      CALL get_molecule_kind(molecule_kind=molecule_kind_b,&
        molecule_list=molecule_list_b, natom=natom_mol_b)
      DO imolecule_b=1,SIZE(molecule_list_b)
        imol_b = imol_b + 1
        DO iatom_mol_b=1,natom_mol_b
          iatom_b = molecule_set(molecule_list_b(imolecule_b))%first_atom +&
            iatom_mol_b - 1
          ! is this block active
          IF (symmetric) THEN
            IF (iatom_a < iatom_b) THEN
              include_ab = .TRUE.
            ELSE
              include_ab = .FALSE.
            END IF
          ELSE
            include_ab = .TRUE.
          END IF
        END DO
      END DO
    END DO
  END DO
END DO
```





All material (C) 2012 by the authors.

This work is licensed under a **Creative Commons Attribution-NonCommercial-ShareAlike 3.0** Unported License, <http://creativecommons.org/licenses/by-nc-sa/3.0/>

Agenda

Tuesday

9.00-9.45	Getting started with Fortran
10.00-10.45	Arrays in Fortran
11.00-12.00	Exercises
12.00-13.00	Lunch break
13.00-13.45	Procedures & modules
14.00-16.00	Exercises

Wednesday

9.00-9.45	Derived datatypes
10.00-10.45	File I/O
11.00-12.00	Exercises
12.00-13.00	Lunch break
13.00-13.45	Exercises
14.00-14.45	Other handy Fortran features
15.00-16.00	Exercises

Thursday

9.00-9.45	Coarray programming model and basic syntax
10.00-11.00	Exercises
11.15-12.00	Further coarray features
12.00-13.00	Lunch break
13.00-14.00	Exercises
14.15-15.00	Advanced coarray features
15.15-16.00	Exercises

Web resources

- CSC's Fortran95/2003 Guide (in Finnish) for free
<http://www.csc.fi/csc/julkaisut/oppaat>
- Fortran wiki: a resource hub for all aspects of Fortran programming
<http://fortranwiki.org>
- GNU Fortran online documents
<http://gcc.gnu.org/onlinedocs/gcc-4.5.0/gfortran>
- Code examples
<http://www.nag.co.uk/nagware/examples.asp>
<http://www.personal.psu.edu/jhm/f90/progref.html>
- Mistakes in Fortran 90 Programs That Might Surprise You
<http://www.cs.rpi.edu/~szymansk/OOF90/bugs.html>

PART I: GETTING STARTED WITH FORTRAN

Outline

- First encounter with Fortran
- Variables and their assignment
- Control structures

Why learn Fortran?

- Well suited for **numerical computations**
 - Likely over 50% of scientific applications are written in Fortran
- **Fast** code (compilers can optimize well)
- Handy **array data types**
- **Clarity** of code
- **Portability** of code
- Optimized **numerical libraries** available

Fortran through the ages

- John W. Backus et al (1954): The IBM Mathematical **Formula Translating System**
- Early years development: Fortran II (1958), Fortran IV (1961), Fortran 66 & Basic Fortran (1966)
- Fortran 77 (1978)
- Fortran 90 (1991) major revision, Fortran 95 (1995) a minor revision to it

Fortran through the ages

- Fortran 2003: major revision, adding e.g. object-oriented features
 - "Fortran 95/2003" is the current *de facto* standard
- The latest standard is Fortran 2008 (approved 2010), a minor upgrade to 2003

Look & Feel

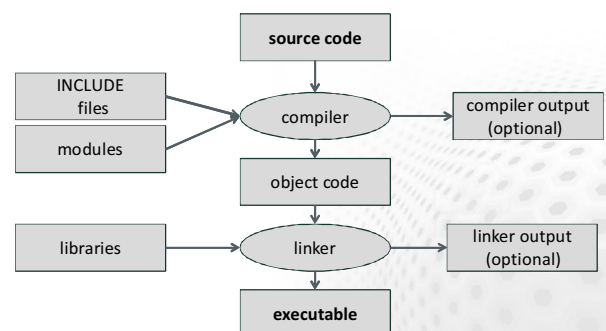
```
PROGRAM square_root_example
! Comments start with an exclamation point.
! You will find data type declarations, couple arithmetic operations
! and an interface that will ask a value for these computations.
IMPLICIT NONE
REAL :: x, y
INTRINSIC SQRT ! Fortran standard provides many commonly used functions

! Command line interface. Ask a number and read it in
WRITE (*,*) 'Give a value (number) for x:'
READ (*,*) x

y=x**2+1 ! Power function and addition arithmetic

WRITE (*,*) 'given value for x:', x
WRITE (*,*) 'computed value of x**2 + 1:', y
! Print the square root of the argument y to screen
WRITE (*,*) 'computed value of SQRT(x**2 + 1):', SQRT(y)
END PROGRAM square_root_example
```

Compiling and linking



Variables

```
IMPLICIT NONE
INTEGER :: n0
REAL :: a, b
REAL :: r1=0.0
COMPLEX :: c
COMPLEX :: imag_number=(0.1, 1.0)
CHARACTER(LEN=80) :: place
CHARACTER(LEN=80) :: name='James Bond'
LOGICAL :: test0 = .TRUE.
LOGICAL :: test1 = .FALSE.
REAL, PARAMETER :: pi=3.14159
```

Variables must be *declared* at the beginning of the program or procedure

The *intrinsic* data types in Fortran are INTEGER, REAL, COMPLEX, CHARACTER and LOGICAL

They can also be given a value at declaration

Constants defined with the PARAMETER clause – they cannot be altered after their declaration

Assignment statements

```
PROGRAM numbers
  IMPLICIT NONE
  INTEGER :: i
  REAL :: r
  COMPLEX :: c, cc
  i = 7
  r = 1.618034
  c = 2.7182818 !same as c = CMPLX(2.7182818)
  cc = r*(1,1)
  CMPLX(r)
  WRITE (*,*) i, r, c, cc
END PROGRAM
```

Automatic change of representation, works between all numeric intrinsic data types

How can I convert numbers to character strings and vice versa? See "INTERNAL I/O" in the File I/O lecture.

Output (one integer and real and two complex values):

```
7 1.618034 (2.718282, 0.000000) (1.618034, 1.618034)
```

Control structures: loops

```
! DO loop with an integer counter (count controlled)
INTEGER :: i, stepsize, NumberOfPoints
INTEGER, PARAMETER :: max_points=100000
REAL :: x_coordinate(max_points), x, totalsum
...
stepsize=2
DO i = 1, NumberOfPoints, stepsize
  x_coordinate(i) = i*stepsize*0.05
END DO
```

```
! Condition controlled loop (DO WHILE)
totalsum = 0.0
READ(*,*) x
DO WHILE (x > 0)
  totalsum = totalsum + x
  READ(*,*) x
END DO
```

Operators

Arithmetic operators

```
REAL :: x,y
INTEGER :: i = 10
x=2.0**(-i) !power function and negation
x=x*REAL(i) !multiplication and type change
x=x/2.0 !division
i=i+1 !addition
i=i-1 !subtraction
```

precedence: first
precedence: second
precedence: second
precedence: third
precedence: third

Relational operators

```
.LT. or < !less than
.LE. or <= !less than or equal to
.EQ. or == !equal to
.NE. or /= !not equal to
.GT. or > !greater than
.GE. or >= !greater than or equal to
```

Logical operators

```
.NOT. !logical negation
.AND. !logical conjunction
.OR. !logical inclusive disjunction
```

precedence: first
precedence: second
precedence: third

Control structures: loops

! DO loop without loop control

```
REAL :: x, totalsum, eps
totalsum = 0.0
DO
  READ(*,*) x
  IF (x < 0) THEN
    EXIT ! exit the loop
  ELSE IF (x > upperlimit) THEN
    CYCLE ! do not execute any statements but
    ! cycle back to the beginning of the loop
  END IF
  totalsum = totalsum + x
END DO
```

Arrays

```
INTEGER, PARAMETER :: M = 100, N = 500
INTEGER :: idx(M)
REAL :: vector(0:N-1)
REAL :: matrix(M, N)
CHARACTER (len = 80) :: screen ( 24)
```

! or

```
INTEGER, DIMENSION(1:M) :: idx
REAL, DIMENSION(0:N-1) :: vector
REAL, DIMENSION(M, N) :: matrix
CHARACTER(len=80), dimension(24) :: screen
```

By default, indexing starts from 1

Control structures: select case

SELECT CASE statements matches the entries of a list against the case index

- Only one found match is allowed
- Usually arguments are character strings or integers
- DEFAULT branch if no match found

```
...
INTEGER :: i
LOGICAL :: is_prime_number,
test_prime_number
...
SELECT CASE (i)
CASE (2,3,5,7)
  is_prime_number = .TRUE.
CASE (1,4,6,8:10)
  is_prime_number = .FALSE.
CASE DEFAULT
  is_prime_number=test_prime_number(i)
END SELECT
...
```

Control structures: conditionals

```
PROGRAM test_if
  IMPLICIT NONE
  REAL :: x,y,eps,t

  WRITE(*,*) 'Give x and y : '
  READ(*,*) x, y
  eps = EPSILON(x)

  IF (ABS(x) > eps) THEN
    t=y/x
  ELSE
    WRITE(*,*) 'division by zero'
    t=0.0
  END IF
  WRITE(*,*) 'y/x = ',t
END PROGRAM
```

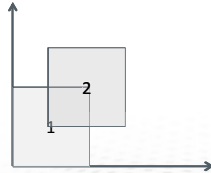
Control structures example

```
PROGRAM gcd
! Computes the greatest common divisor, Euclidean algorithm
IMPLICIT NONE
INTEGER :: m, n, t
WRITE(*,*) 'Give positive integers m and n : '
READ(*,*) m, n
WRITE(*,*) 'm:', m, ' n:', n
positive_check: IF (m > 0 .AND. n > 0) THEN
  main_algorithm: DO WHILE (n /= 0)
    t = MOD(m,n)
    m = n
    n = t
  END DO main_algorithm
  WRITE(*,*) 'Greatest common divisor: ',m
ELSE
  WRITE(*,*) 'Negative value entered'
END IF positive_check
END PROGRAM gcd
```

Labels can be given to control structures and used in conjunction with e.g. exit and cycle statements

Another example

```
PROGRAM placetest
  IMPLICIT NONE
  LOGICAL :: in_square1, in_square2
  REAL :: x,y
```



```
  WRITE(*,*) 'Give point coordinates x and y'
  READ (*,*) x, y
  in_square1 = (x >= 0. .AND. x <= 2. .AND. y >= 0. .AND. y <= 2.)
  in_square2 = (x >= 1. .AND. x <= 3. .AND. y >= 1. .AND. y <= 3.)
  IF (in_square1 .AND. in_square2) THEN ! inside both
    WRITE(*,*) 'Point within both squares'
  ELSE IF (in_square1) THEN ! inside square 1 only
    WRITE(*,*) 'Point inside square 1'
  ELSE IF (in_square2) THEN ! inside square 2 only
    WRITE(*,*) 'Point inside square 2'
  ELSE ! both are .FALSE.
    WRITE(*,*) 'Point outside both squares'
  END IF
END PROGRAM placetest
```

Source code remarks

- A variable name can be no longer than 31 characters (containing only letters, digits or underscore, must start with a letter)
- Maximum row length is 132 characters
- There can be max 39 continuation lines
 - if a line is ended with ampersand (&), the line continues onto the next line.
- No distinction between lower and uppercase characters
 - Character strings are case sensitive

Source code remarks

```
! Character strings are case sensitive
CHARACTER(LEN=32) :: ch1, ch2
Logical :: ans
ch1 = 'a'
ch2 = 'A'
ans = ch1 .EQ. ch2
WRITE(*,*) ans ! OUTPUT from that WRITE statement is: F

! When strings are compared
! the shorter string is extended with blanks
WRITE(*,*) 'A' .EQ. 'A ' !OUTPUT: T
WRITE(*,*) 'A' .EQ. ' A' !OUTPUT: F

! Statement separation: newline and semicolon, ;
! Semicolon as a statement separator
a = a * b; c = d**a
! The above is equivalent to following two lines
a = a * b
c = d**a
```

Summary

- Fortran 95/2003 is – despite its long history - a modern programming language especially for scientific computing
 - Versatile, quite easy to learn, powerful
- In our first encounter, we discussed
 - Variables & data types
 - Control structures: loops & conditionals
 - Operators



PART II: FORTRAN ARRAYS

Outline

- Significance of Fortran arrays
- Array syntax & array sections
- Dynamic memory allocation
- Array intrinsic functions

Significance of Fortran arrays

- *Fortran arrays* enable a natural and versatile way to access multi-dimensional data during computation
 - Matrices, vectors,...
 - Array has particular data type (same for all elements)
 - *Dimension* specified in the variable declaration
 - Fortran supports up to 7 dimensional arrays

Array declaration

```
INTEGER, PARAMETER :: M = 100, N = 500
INTEGER :: idx(M), lms(-2:2)
REAL :: matrix(M, N)

! or

INTEGER, DIMENSION(M) :: idx, idx2, idx3
```

- Indexing starts from 1 by default
- Different indexing can be defined in declaration

Array syntax

- In older Fortran, arrays were traditionally accessed element-by-element basis
- Modern Fortran has a way of accessing several elements in one go → *array syntax*

$$y(:) = y(:) + A(:, J) * x(J)$$

- Array syntax potentially improves code readability
- Array syntax may also give the Fortran compiler a chance for better performance optimization

Array syntax

- Array syntax allows for less explicit DO loops

```
INTEGER, PARAMETER :: M = 4, N = 5
REAL (kind = 8) :: A(M,N), x(N), y(M)
INTEGER :: I, J

y(:) = 0

OUTER_LOOP : do J = 1, N
  INNER_LOOP : do I = 1, M
    y(:) = y(:) + A(:, J) * x(J)
  end do INNER_LOOP
end do OUTER_LOOP
```

Array syntax

- Element-by-element initialization

```
do j = 0, 10
  vector (j) = 0
  idx (j) = j
end do
```
- Using array syntax in initialization

```
vector = 0
! or
vector(:) = 0

idx(0:10) = (/ (j, j = 0, 10) /)
```

Array sections

- With Fortran array syntax we can access a part of an array in a pretty intuitive way: *array sections*

```
Sub_Vector ( 3:N+8) = 0
Every_Third ( 1:3*N+1 : 3 ) = 1
Diag_Block ( i-1:i+1, j-2:j+2 ) = k
```
- Sections enable us to refer to (say) a sub-block of a matrix, or a sub-cube of a 3D-array:

```
REAL(kind = 8) :: A ( 1000, 1000)
INTEGER (kind = 2) :: pixel_3D(256, 256, 256)
A(2:500, 3:300:3) = 4.0
pixel_3D (128:150, 56:80, 1:256:8) = 32000
```

Array sections

- Be aware of: when copying array sections, then both left and right hand sides of the assignment statement has to have conforming dimensions :

```
LHS(1:3, 0:9) = RHS(-2:0, 20:29) ! This is OK  
! but here is an error :  
LHS(1:2, 0:9) = RHS(-2:0, 20:29)
```

Memory allocation with automatic arrays

```
SUBROUTINE CALCULATE(M, N)  
  INTEGER, INTENT(IN) :: M, N ! Intended dimensions of the arrays  
  
  INTEGER :: idx(0:M-1)      ! An automatic array  
  REAL(kind = 8) :: mat(M,N) ! An automatic array  
  
  ! No explicit ALLOCATE ☹ - but no checks upon failure either ☹  
  
  ! Use idx & mat automatic arrays here  
  CALL DO_SOMETHING(M, N, idx, mat)  
  
  ! No explicit DEALLOCATE ☹ - memory gets reclaimed automatically  
  
END SUBROUTINE CALCULATE
```

Dynamic memory allocation

- So far in our examples the array dimensions have been declared at compile time:
 - memory allocation is *static*
- If an array size depends on the input to program, its memory should be allocated at runtime:
 - memory allocation becomes *dynamic*

Array intrinsic functions

- Built-in functions can apply various operations on whole array, not just array elements
- As a result either another array or just a scalar value is returned
- A subset selection through *masking* is possible
 - Masking and use of array (intrinsic) functions is often accompanied with use of FORALL and WHERE array statements

Dynamic memory allocation

- Fortran provides two different mechanisms to allocate memory dynamically through arrays:
 - Array variable declaration has an ALLOCATABLE (or a POINTER) attribute, and memory is allocated through the ALLOCATE statement, and freed through DEALLOCATE
 - A variable, which is declared in the procedure with size information coming from the argument list or from a module, is an *automatic array* - no ALLOCATE is needed, neither DEALLOCATE

Array intrinsic functions

- SIZE** (array [, dim]) returns # of elements in the array, optionally along the specified dimension
- SHAPE** (array) returns an INTEGER vector containing SIZE of array with respect to each of its dimension
- COUNT** (L_array [,dim]) returns the count of elements which are .TRUE. in the LOGICAL L_array
- SUM** (array[, dim][, mask]) : sum of the elements, optionally along a dimension, and optionally under mask

Dynamic memory allocation

```
INTEGER :: M, N, alloc_stat  
INTEGER, ALLOCATABLE :: idx(:)  
REAL(kind = 8), ALLOCATABLE :: mat(:, :)  
M = 100  
N = 200  
ALLOCATE ( idx (0:M-1) , STAT=alloc_stat )  
IF (alloc_stat /= 0) CALL abort()  
  
ALLOCATE ( mat (M,N) , STAT=alloc_stat )  
IF (alloc_stat /= 0) CALL abort()  
...  
DEALLOCATE (idx , mat)
```

Please do remember a DEALLOCATE for every ALLOCATE statement used !!

Array intrinsic functions

- ANY** (L_array [, dim]) returns a scalar value of .TRUE. if any value in LOGICAL L_array is found to be .TRUE.
- ALL** (L_array [, dim]) returns a scalar value of .TRUE. if all values in LOGICAL L_array are .TRUE.
- MINVAL** / **MAXVAL** (array [,dim] [, mask]) return the minimum/maximum value in a given array [along specified dimension] [, under mask]
- MINLOC** / **MAXLOC** (array [, mask]) return a vector of location(s) [, under mask], where the minimum/maximum value(s) is/are found

Array intrinsic functions

```

INTEGER :: M, N
REAL :: X(M,N), V(N)
PRINT *,SIZE(X), SIZE(V) ! M, N, N
PRINT *,SHAPE(X) ! M, N
PRINT *,SIZE(SHAPE(X)) ! 2
PRINT *,COUNT(X >= 0)
PRINT *,SUM(X, DIM=2, MASK=X < 1)

PRINT *,ANY(V > -1 .and. V < 1)
PRINT *,ALL(X >= 0, DIM=1)
PRINT *,MINVAL(V), MAXVAL(V)
PRINT *,MINLOC(V), MAXLOC(V)

```

Array intrinsic functions

Examples of array control statements

```

INTEGER :: j, ix(5)
ix(:) = (/ (j, j=1,size(ix)) /)

WHERE (ix == 0) ix = -9999

WHERE (ix < 0)
  ix = -ix
ELSEWHERE
  ix = 0
END WHERE

INTEGER :: j
REAL :: a(100,100), b(100), c(100)

! Fill in diagonal matrix
FORALL (j=1:100) a(j,j) = b(j)

! Fill in lower bi-diagonal matrix
FORALL (j=2:100) a(j,j-1) = c(j)

```

Array intrinsic functions

- **RESHAPE** (array, shape) returns a reconstructed array with different shape than in the input array, for example:
 - Can be used as a single line statement to initialize an array (often in expense of readability)
 - Create from an M-by-N matrix a vector of length MxN

```

INTEGER :: M, N
REAL :: A(M, N), V(M*N)
! "Carbon"-copy A to V without loops
V = RESHAPE(A, SHAPE(V))

```

Summary

- Arrays make Fortran language a very versatile vehicle for computationally intensive program development
- Using its *array syntax*, vectors and matrices can be initialized and used in a very intuitive way
- *Dynamic memory allocation* enables sizing of arrays according to particular needs
- *Array intrinsic functions* further simplify coding effort and improve code readability

Array intrinsic functions

- Some array functions manipulate vectors/matrices effectively :
 - **DOT_PRODUCT** (a, b) returns a dot product of two vectors
 - **MATMUL** (a, b) returns matrix multiply of two matrices
 - **TRANSPOSE** (a) returns transposed of the input matrix

```

INTEGER :: L, M, N
REAL :: A(L,M), B(M,N), C(L,N)
REAL :: A_tr(M,L)
REAL :: V1(N), V2(N), DOTP

A_tr = TRANSPOSE(A)
C = MATMUL(A, B)
DOTP = DOT_PRODUCT(V1, V2)

```

Array intrinsic functions

- Array control statements **FORALL** and **WHERE** are commonly used in the context of manipulating arrays
 - These are frankly speaking not array intrinsic functions, but very closely related to
- They can provide a masked assignment of values using effective vector operations



PART III: PROCEDURES & MODULES

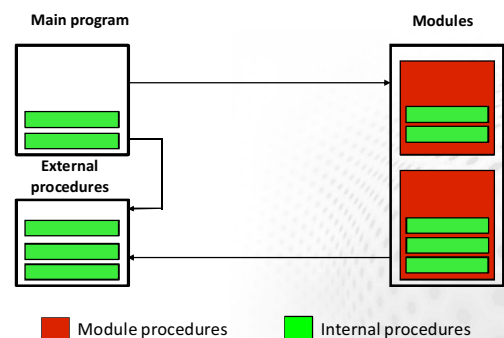
Outline

- Structured programming
- Procedures: functions and subroutines
- Procedure arguments
- Modules

Procedure types

- *Internal, external, and module procedures*
 - Internal procedure: within the program structure
 - External procedure: independently declared, may be on other language
 - Module procedure: defined in a *module*
- Internal and module procedures provide a defined interface, compiler uses this to check arguments
- No type conversions will occur during the procedure call: the data type of an argument on the call site and in the procedure must match

Program units



Structured programming

- Structured programming based on program sub-units (*functions, subroutines and modules*) enables
 - testing and debugging separately
 - re-use of code
 - improved readability
 - re-occurring tasks

Declaration

Function

```
[TYPE] FUNCTION func(ARGS)
[RESULT(arg)]
  [declarations]
  [statements]
END FUNCTION func
```

- Call convention
- ```
res = func(ARGS)
```

#### Subroutine

```
SUBROUTINE sub(arguments)
 [declarations]
 [statements]
END SUBROUTINE sub
```

- Call convention
- ```
CALL sub(arguments)
```

What are procedures?

- With procedures we mean *subroutines* and *functions*
- Subroutines exchange data through its argument lists only
- Functions return a value according to its declared data type

Declaration

```
INTEGER FUNCTION test(s)
  IMPLICIT NONE
  INTEGER :: s
  test=10*s
END FUNCTION test
```

```
PROGRAM do_something
...
result=test(s)
...
```

```
SUBROUTINE test(s,test)
  IMPLICIT NONE
  INTEGER :: s, test
  test=10*s
END SUBROUTINE test
```

```
PROGRAM do_something
...
call test(s,result)
...
```

Internal procedures

- Each program unit (program/subroutine/function) may contain *internal procedures*
- Declared at the end of a program unit after the CONTAINS statement
 - Nested CONTAINS statements not allowed
- Inherits variables and objects from the program unit

Procedure arguments

- We may pass into procedures also *other procedures* (i.e., not only data)
- Internal procedures cannot be used as arguments

```
PROGRAM degtest
  IMPLICIT NONE
  INTRINSIC ASIN, ACOS, ATAN
  WRITE (*,*) 'arcsin(0.5): ', deg(ASIN,0.5)
  WRITE (*,*) 'arccos(0.5): ', deg(ACOS,0.5)
  WRITE (*,*) 'arctan(1.0): ', deg(ATAN,1.0)
CONTAINS
  REAL FUNCTION deg(f, x)
    IMPLICIT NONE
    INTRINSIC ATAN
    REAL, EXTERNAL :: f
    REAL, INTENT(IN) :: x
    deg = 45*f(x)/ATAN(1.0)
  END FUNCTION deg
END PROGRAM degtest
```

External procedures

- Declared in a separate program unit
 - Referred to with the EXTERNAL keyword
 - Have to be compiled separately and linked
- Do not use them within a program: modules are much easier and more robust
- External procedures needed when using
 - procedures written with different programming language
 - library routines (e.g. BLAS)

Passing array arguments

- Two ways to pass arrays to procedures
 - Explicit shape array (dimensions passed explicitly)
subroutine foo(size1, size2, ..., matrix, ...)
...
real, dimension(size1,size2) :: matrix
 - Assumed (implicit) shape array
real, dimension(:,) :: matrix

Procedure arguments

- Call by reference: any change to argument's value changes the actual argument
- Behavior of the arguments can be controlled with the INTENT keyword
- Compiler checks the arguments if the interface of procedure is known at compilation time (in internal and module procedures)
 - For external routines it can be declared with the INTERFACE block

Interface definition

- It is a good practice to define the interfaces for external procedures
 - Enables compilation time error checking
- The INTERFACE block can be also used for defining sc. generic procedures

```
SUBROUTINE nag_rand(table)
  INTERFACE
    SUBROUTINE g05faf(a, b, n, x)
      REAL, INTENT(IN) :: a
      REAL, INTENT(IN) :: b
      INTEGER, INTENT(IN) :: n
      REAL, INTENT(OUT), DIMENSION(n) :: x
    END SUBROUTINE g05faf
  END INTERFACE
  REAL, DIMENSION(:), INTENT(OUT) :: table
  CALL g05faf(-1.0, 1.0, SIZE(table), table)
END SUBROUTINE nag_rand
```

Defining an interface for the g05faf subroutine of the NAG library (generates a set of random numbers)

INTENT keyword

- Declares how formal argument is intended for transferring a value
 - in
 - out
 - inout (default)
- Compiler uses INTENT for error checking and optimization

```
SUBROUTINE foo(x,y,z)
  IMPLICIT NONE
  REAL,INTENT(in) :: x
  REAL,INTENT(inout) :: y
  REAL,INTENT(out) :: z

  x=10 ! Compilation error
  y=10 ! Correct
  z=y*x ! Correct
END SUBROUTINE foo
```

Modular programming

- Modularity means dividing a program into small minimally dependent *modules*
- Advantages
 - Constants, variables, data types and procedures can be defined in modules
 - Makes it possible to divide program into smaller self-contained units

Usefulness of Fortran modules

- Global definitions
 - The same procedures and data types available in different program units
- Compile-time error checks
- Hide implementation details (object oriented programming)
- Group routines and data structures
- Define generic procedures and custom operators

Visibility of objects

- Variables and procedures in modules can be PRIVATE or PUBLIC
 - PUBLIC = visible for all program units using the module (default)
 - PRIVATE will hide the objects from other program units
- ```
INTEGER, PRIVATE :: x
INTEGER, PUBLIC :: y
```

## Module procedures

- Procedures defined in modules can be used in any other program unit
- Placing procedures in modules helps compiler to detect programming errors and to optimize the code
- Module procedures are declared after the CONTAINS statement

## Summary

- Procedural programming makes the code more readable and easier to modify
  - Procedures encapsulate some piece of work that makes sense and may be worth re-using elsewhere
  - Fortran uses *functions* and *subroutines*
  - Values of procedure arguments may be changed upon calling the procedure
- Fortran *modules* are used for modular programming and data encapsulation

## Using modules

### Declaration

```
MODULE check
 IMPLICIT NONE
 INTEGER, PARAMETER :: &
 intp = SELECTED_INT_KIND(8)
CONTAINS
 FUNCTION check_this(x) RESULT(z)
 INTEGER(intp):: x, z
 ...
 END FUNCTION
END MODULE check
```

### Usage

```
PROGRAM testprog
 USE check
 IMPLICIT NONE
 INTEGER(KIND=intp) :: x, test
 test=check_this(x)
END PROGRAM testprog
```

• A good habit

```
USE accuracy, ONLY: realp
```

## Common variables with modules

- In many Fortran 77 codes, so called *common blocks* are used
  - Referenced and/or modified anywhere in the program

```
COMMON/EQ/N,NTOT
COMMON/TOL/ABSTOL,RELTOL
```
- The Fortran 95/2003 way to do it is with modules

```
MODULE commons
 INTEGER, PARAMETER :: r = 0.42
 INTEGER, SAVE :: n, ntot
 REAL, SAVE :: abstol, reltol
END MODULE commons
```



## PART IV: DERIVED DATA TYPES

### Numerical precision

```
PROGRAM Precision_Test
IMPLICIT NONE

INTEGER, PARAMETER :: sp = SELECTED_REAL_KIND(6,30), &
 dp = SELECTED_REAL_KIND(10,200)

REAL(KIND=sp) :: a
REAL(KIND=dp) :: b
WRITE(*,*) sp, dp, KIND(1.0), KIND(1.0_dp)
WRITE(*,*) KIND(a), HUGE(a), TINY(a), RANGE(a), PRECISION(a)
WRITE(*,*) KIND(b), HUGE(b), TINY(b), RANGE(b), PRECISION(b)

END PROGRAM Precision_Test
```

Output:

```
4 8 4 8
4 3.4028235E+38 1.1754944E-38 37 6
8 1.797693134862316E+308 2.225073858507201E-308 307 15
```

### Outline

- Recalling Fortran built-in data types
- Rationale behind derived data types
- Data type declaration and visibility with examples

### Numerical precision

Other intrinsic functions related to numerical precision

|                |                                                                                  |
|----------------|----------------------------------------------------------------------------------|
| KIND(A)        | Returns the kind of the supplied argument                                        |
| TINY(A)        | The smallest positive number                                                     |
| HUGE(A)        | The largest positive number                                                      |
| EPSILON(A)     | The smallest positive number added to 1.0 returns a number just greater than 1.0 |
| PRECISION(A)   | Decimal precision                                                                |
| DIGITS(A)      | Number of significant digits                                                     |
| RANGE(A)       | Decimal exponent                                                                 |
| MAXEXPONENT(A) | Largest exponent (of the kind(A))                                                |
| MINEXPONENT(A) | Smallest exponent (of the kind(A))                                               |

### Fortran built-in types

- Standard Fortran already supports a wide variety of fundamental data types to represent integers, floating point numbers (real), truth values (logical) and variable length character strings
- In addition each of these built-in types may have declared as multi-dimensional array
- Furthermore, reals and integers can be declared to consume less memory in expense of reduced numerical precision through kind parameter (e.g. 8 or 4)

### What is derived data type ?

- Derived data type is a data structure which is defined by the programmer
  - Equivalent to structs in C programming language
  - Consists of other data types – including other derived types
- Derived type is defined in the variable declaration section of programming unit
  - Not visible to other programming units
    - Unless defined in modules and used via USE clause

### A few words about numerical precision

- The variable representation method (precision) may be declared using the KIND statement

```
! SELECTED_INT_KIND(r)
! SELECTED_REAL_KIND(p)
! SELECTED_REAL_KIND(p,r)
```

Integer between  $-10^r < n < 10^r$

Real number accurate to  $p$  decimals

```
INTEGER, PARAMETER :: short=SELECTED_INT_KIND(4)
INTEGER, PARAMETER :: double=SELECTED_REAL_KIND(12,100)
INTEGER (KIND=short) :: index
REAL (KIND=double) :: x,y,z
COMPLEX (KIND=double) :: c
```

```
x=1.0_double; y=2.0_double * ACOS(x)
```

A real number between  $-10^{100} < x < 10^{100}$ , accurate to 12 decimals

### Derived data types – rationale

- A well structured and readable program should try to express its variables as objects
- Variables used in the same context should be grouped together – this makes the program flow easier to understand and less prone to errors
- Derived data types is a feature which allows programmer to make object grouping possible

## Data type declaration

- Type declaration:

```
TYPE playertype
 CHARACTER (LEN=30) :: name
 INTEGER :: number
 REAL :: rating
END TYPE playertype
```

- Declaring variables using a derived data type:

```
TYPE(playertype) :: john, luiz
TYPE(playertype), DIMENSION(10) :: players
```

## Visibility of derived data types

- When declared in a module unit, a derived data type can be accessed outside the module through USE-statement:

```
MODULE TYPEMOD
 TYPE MY_TYPE
 INTEGER :: FOO
 END TYPE MY_TYPE
PUBLIC :: MY_TYPE
END MODULE TYPEMOD

SUBROUTINE SUB
 USE TYPEMOD, ONLY : MY_TYPE
 TYPE(MY_TYPE) :: VAR
 VAR%FOO = 7
 CALL CALC (VAR)
END SUBROUTINE SUB

SUBROUTINE CALC (NEWVAR)
 USE TYPEMOD, ONLY : MY_TYPE
 TYPE(MY_TYPE) :: NEWVAR
 PRINT *,NEWVAR
END SUBROUTINE CALC
```

## Accessing data types

- Initialization

```
john%name = 'John'
john%number = 11
john%rating = 7.3
```

- Alternatively

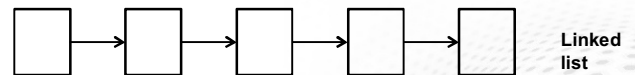
```
Luiz = playertype('Luiz', 8, 6.4)
```

- Vector of derived data type: element-wise addressing

```
players(1)%name = 'Phil'
players(1)%number = 4
players(1)%rating = 5.5
```

## A linked list

- A chain of data elements in a dynamically extendable list



- For example

```
TYPE list_of_teams
 TYPE(footballteam) :: team
 TYPE(list_of_teams), POINTER :: next_team => NULL()
END TYPE list_of_teams
```

## Nested data types

- Declaration of a data type using another data type:

```
TYPE footballteam
 CHARACTER (LEN=80) :: name
 TYPE(playertype) :: player(11)
END TYPE footballteam
```

- Declaring variables:

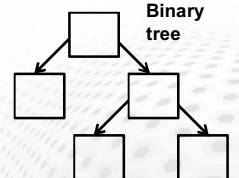
```
TYPE(footballteam) :: manu, arsenal, reading
```

- Initialization / access example:

```
manu%name = 'Manchester United'
manu%player(1)%name = 'Robin van Persie'
manu%player(1)%number = 20
```

## A binary tree

- Data elements in a binary tree offer superb search speed in just  $O(N \log_2 N)$  operations



- Consists of left and right leafs

```
TYPE binary_tree
 TYPE(my_data_type) :: data
 TYPE(binary_tree), POINTER :: left => NULL()
 TYPE(binary_tree), POINTER :: right => NULL()
END TYPE binary_tree
```

## Visibility of derived data types

- When declared in the same programming unit (variable declaration section) derived data types are visible to that unit only (and sub-units under CONTAINS statement) :

```
SUBROUTINE SUB
 ...
 TYPE MY_TYPE
 INTEGER :: FOO
 END TYPE MY_TYPE

 TYPE(MY_TYPE) :: VAR
 VAR%FOO = 7
 CALL CALC
CONTAINS
 ...
 SUBROUTINE CALC
 TYPE(MY_TYPE) :: NEWVAR
 NEWVAR = VAR
 PRINT *,NEWVAR
 END SUBROUTINE CALC
END SUBROUTINE SUB
```

## Summary

- Derived data types enables grouping of data to form logical objects
- A Fortran program becomes more readable and modular with sensible use of derived data types
- Handling of linked lists or binary trees becomes more manageable with use of data structures

## PART V: FILE I/O

### Outline

- Input/output formatting
- File opening and closing
- Writing and reading to/from a file
- Formatted and unformatted files
- Stream I/O
- Internal I/O

### Input/Output formatting

- To prettify output and to make it human readable, use FORMAT descriptors in connection with the WRITE statement
- Although less often used nowadays, it can also be used with READ to input data at fixed line positions and using predefined field lengths
- Use either through FORMAT statements, CHARACTER variable or embedded in READ / WRITE fmt keyword

### Output formatting

| Data type                                          | FORMAT descriptors              | Examples                                                                  |
|----------------------------------------------------|---------------------------------|---------------------------------------------------------------------------|
| Integer                                            | Iw, Iw.m                        | WRITE(*,'(I5)') J<br>WRITE(*,'(I5.3)') J<br>WRITE(*,'(I0)') J             |
| Real (decimal and exponential forms, auto-scaling) | Fw.d<br>Ew.d, Ew.dEe<br>1P,Gw.d | WRITE(*,'(F7.4)') R<br>WRITE(*,'(E12.3E4)') R<br>WRITE(*,'(1P,G20.13)') R |
| Character                                          | A, Aw                           | WRITE(*,'(A)') C                                                          |
| Logical                                            | Lw                              | WRITE(*,'(L2)') L                                                         |

w=width of the output field, d=number of digits to the right of decimal point, m=minimum number of characters to be used, e=number of digits in the exponent. Variables: Integer :: J, Real :: R, Character :: C, Logical :: L

### Output formatting: miscellaneous

- With complex numbers provide format for both real and imaginary parts:  

```
COMPLEX :: Z
WRITE (*,'(F6.3,2X,F6.3)') Z
```
- Line break and tabbing:  

```
WRITE (*,'(F6.3,/,F6.3)') X, Y
WRITE (*,'(I5,T20,I5)') I, J
```
- It is possible that an edit descriptor will be repeated a specified number of times  

```
WRITE (*,'(5I8)')
WRITE (*,'(3(I5,F8.3))')
```

### Opening & closing files : basic concepts

- Writing to or reading from a file is similar to writing onto a terminal screen or reading from a keyboard
- Differences
  - File must be opened with an OPEN statement, in which the unit number and (optionally) the file name are given
  - Subsequent writes (or reads) must refer to the given unit number
  - File should be closed at the end

### Opening & closing a file

- The syntax is (the brackets [ ] indicate optional keywords or arguments) :  

```
OPEN([unit=]iu, file='name' [, options])
CLOSE([unit=]iu [, options])
```
- For example :  

```
OPEN(10, file= 'output.dat', status='new')
CLOSE(unit=10, status='keep')
```

### Opening & closing a file

- The first parameter is the unit number
- The keyword unit= can be omitted
- The unit numbers 0, 5 and 6 are predefined
  - 0 is output for standard (system) error messages
  - 5 is for standard (user) input
  - 6 is for standard (user) output
  - These units are opened by default and should not be re-opened nor closed by the user

## Opening & closing a file

- The default output/input unit can be referred with a star:

```
WRITE(*, ...)
READ(*, ...)
```

- Note that these are *not* necessarily the same as the stdout and stdin unit numbers 6 and 5
- If the file name is omitted in the OPEN, the file name will be based on unit number being opened, e.g. for unit=12 this usually means the filename 'fort.12' (on Unix/MAC)

## File opening options

- **status** : existence of the file
  - 'old', 'new', 'replace', 'scratch', 'unknown'
- **position** : offset, where to start writing
  - 'append'
- **action** : file operation mode
  - 'write', 'read', 'readwrite'
- **form** : text or binary file
  - 'formatted', 'unformatted'

## File opening options

- **access** : direct or sequential file access
  - 'direct', 'sequential', 'stream',
- **iostat** : error indicator, (output) integer
  - Non-zero only upon an error
- **err** : the label number to jump upon error
- **recl** : record length, (input) integer
  - For direct access files only
  - Warning (check): may be in bytes or words

## File opening: file properties

- Use INQUIRE statement to find out information about
  - file existence
  - file unit open status
  - various file attributes
- The syntax has two forms, one based on file name, the other for unit number

```
INQUIRE(file='name', options ...)
INQUIRE(unit=iu, options ...)
```

## File opening: file properties

- **exist** : does file exist ? (LOGICAL)
- **opened** : is file / unit opened ? (LOGICAL)
- **form** : 'formatted' or 'unformatted' (CHAR)
- **access** : 'sequential' or 'direct' or 'stream' (CHAR)
- **action** : 'read', 'write', 'readwrite' (CHAR)
- **recl** : record length (INTEGER)
- **size** : file size in bytes (INTEGER)

## File opening: file properties

- Find out about existence of a file

```
LOGICAL :: file_exist

INQUIRE (FILE='foo.dat', EXIST=file_exist)
IF (.NOT. file_exist) THEN
 WRITE(*,*) 'The file does not exist'
ELSE
 ! Do something with the file 'foo.dat'
ENDIF
```

## File writing and reading

- Writing to and reading from a file is done by giving the corresponding unit number (iu) as a parameter :

```
WRITE(iu,*) str
WRITE(unit=iu, fmt=*) str
READ(iu,*) str
READ(unit=iu, fmt=*) str
```

The star format (\*) indicates list-directed output (i.e. programmer does not choose the input/output styles)

- Formats and other options can be used as needed
- If keyword 'unit' used, also 'fmt' keyword must be used ('fmt' is applicable to formatted, text files only)

## Formatted vs. unformatted files

- Text or *formatted* files are
  - Human readable
  - Portable i.e. machine independent
- Binary or *unformatted* files are
  - Machine readable only, *not* portable
  - Much *faster to access* than formatted files
  - Suitable for large amount of data due to *reduced file sizes*
  - Internal data representation used for numbers, thus no number conversion, no rounding of errors compared to formatted data

## Unformatted I/O

- Write to a sequential binary file

```
REAL rval
CHARACTER(len = 60) string
OPEN(10,file='foo.dat',form='unformatted')
WRITE(10) rval
WRITE(10) string
CLOSE(10)
```

- No FORMAT descriptors allowed

- Reading similarly

```
READ(10) rval
READ(10) string
```

## Summary

- Input/Output formatting
- Files: communication between a program and the outside world
  - Opening and closing a file
  - Data reading & writing
- Use unformatted (binary) I/O for all except text files
- Stream I/O
- Internal I/O

## Stream I/O

- A binary file write adds extra record delimiters (hidden from programmer) to the beginning and end of records
- In Fortran2003 using access method 'stream' avoids this and implements (finally!) a C-programming like approach

- Create a stream (binary) file

```
REAL dbheader(20), dbdata(300)
OPEN(10,file='my_database.dat',access='stream')
WRITE(10) dbheader
WRITE(10) dbdata
CLOSE(10)
```

- Reading similarly

## Internal I/O

- Often it is necessary to filter out data from a given character string
- Or to pack values into a character string
- Fortran internal I/O with READ & WRITE becomes handy
- No actual (physical) files are involved at all

## Internal I/O : Examples

- Extract a number from a given character string :

```
CHARACTER(LEN=13) :: CL = 'Time step# 10'
INTEGER :: ISTEP
READ(CL,fmt='(10X,I3)') ISTEP
```

- Write text and data to a character string:

```
CHARACTER(LEN=60) CL
WRITE(CL,fmt='(A,I0)') 'The number of jobs completed = ',100
```



## PART VI: OTHER HANDY FORTRAN 95/2003 FEATURES

### Outline

- Pointer aliasing
- Generic procedures
- Special procedure attributes & optional procedure attributes
- Command-line arguments

## Generic procedures

- Procedures which perform similar actions but for different data types can be defined as **generic procedures**
- Procedures are called using the **generic name** and compiler uses the correct procedure based on the argument number, type and dimensions
  - Compare with "overloading" in C++
- Generic name is defined in INTERFACE section

### Generic procedures example

```

MODULE swapmod
 IMPLICIT NONE
 INTERFACE swap
 MODULE PROCEDURE swap_real, swap_char
 END INTERFACE
CONTAINS
 SUBROUTINE swap_real(a, b)
 REAL, INTENT(INOUT) :: a, b
 REAL :: temp
 temp = a; a = b; b = temp
 END SUBROUTINE
 SUBROUTINE swap_char(a, b)
 CHARACTER, INTENT(INOUT) :: a, b
 CHARACTER :: temp
 temp = a; a = b; b = temp
 END SUBROUTINE
END MODULE swapmod

```

```

PROGRAM switch
 USE swapmod
 IMPLICIT NONE
 CHARACTER :: n,s
 REAL :: x,y
 n = 'J'
 s = 'S'
 x=10
 y=20
 PRINT *,x,y
 PRINT *,n,s
 CALL swap(n,s)
 CALL swap(x,y)
 PRINT *,x,y
 PRINT *,n,s
END PROGRAM

```

### Pointers to arrays

- The POINTER attribute enables to create array (or scalar) **aliasing variables**
- Pointer variables are usually employed to **refer** to another array or array section
- A pointer variable can also be a sole variable itself, but requires ALLOCATE
  - This is not a recommended practice – use the ALLOCATABLE attribute and employ POINTERS for aliasing only
- C programmers: a "pointer" has a slightly different meaning in C and Fortran

### Special attributes for procedures: RECURSIVE

- Recursion means calling a procedure within itself
- Triggered via RECURSIVE keyword

```

RECURSIVE FUNCTION factorial(n) RESULT(fac)
 INTEGER, INTENT(IN) :: n
 INTEGER :: fac
 IF (n==0) THEN
 fac=1
 ELSE
 fac=n*factorial(n-1)
 END IF
END FUNCTION factorial

```

### Pointers to arrays

- A POINTER can *refer* to an already allocated memory region

```

INTEGER, POINTER :: p_x(:) => NULL()
INTEGER, TARGET :: x(1000)
...
p_x => x
p_x => x (2 : 300)
p_x => x (1 : 1000 : 5)
...
p_x(1) = 0
NULLIFY(p_x)

```

Initialized to point to nothing

Pointers provide a neat way for array sections

This would change also x(1) to 0

Disconnects p\_x's connection to x

### Special attributes for procedures: PURE

- PURE keyword indicates that the function is free of side effects
  - Such as a change in value of an input argument or global variable
- Intrinsic functions are always pure
- No (external) I/O is allowed in PURE procedures
- Pure procedure must specify intents of its all arguments
- The motivation is **efficiency**: Enables more aggressive compiler optimization and parallelization with e.g. OpenMP

## Special attributes for procedures: ELEMENTAL

- The ELEMENTAL attribute allows for declaring procedures that operate element-by-element
- The argument can be a scalar or an array of any shape
- This is another way for defining more general procedures

```
ELEMENTAL REAL FUNCTION f(x,y)
 REAL, INTENT(IN) :: x, y
 f = SQRT(x**2 + y**2)
END FUNCTION f

...
REAL, DIMENSION(n,n) :: a, b, c
REAL, DIMENSION(n) :: t, u, v
...
c = f(a,b)
v = f(t,u)
```

## Command line input

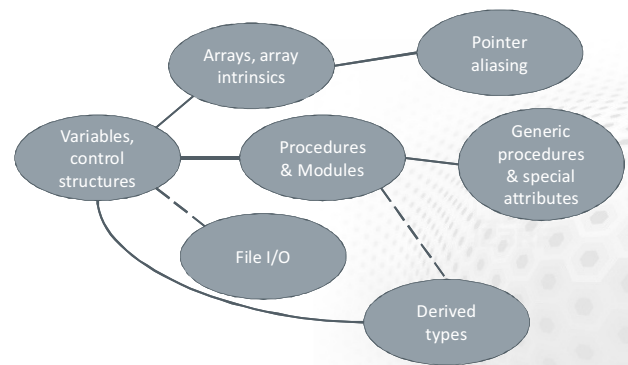
- Example: reading in two integer values from the command line
- The (full) program should be launched as (e.g.)  
% ./a.out 100 100

```
subroutine read_command_line(height, width)
 integer, intent(out) :: height, width
 character(len=10) :: args(2)
 integer :: n_args, i
 n_args = command_argument_count()
 if (n_args /= 2) then
 write(*,*) ' Usage : ./exe height width '
 call abort()
 end if
 do i = 1, 2
 call get_command_argument(i,args(i))
 args(i) = trim(adjustl(args(i)))
 end do
 read(args(1),*) height
 read(args(2),*) width
end subroutine read_command_line
```

## SAVEd variables

- By default objects in procedures are dynamically allocated upon invocation
- Only saved variables keep their value from one call to the next
  - SAVE attribute  
REAL, **SAVE** :: a
  - Variables assigned with a value upon declaration are equal to SAVE attribute (C programmers should note this!)  
REAL :: a = 1.0

## Fortran 95/2003 crash course summary



## Optional procedure arguments

- Procedure arguments can be defined as *optional*, i.e., using some predefined value for arguments not provided

```
REAL FUNCTION average(x, low, up)
 IMPLICIT NONE
 REAL, DIMENSION(:), INTENT(IN) :: x
 REAL, INTENT(IN), OPTIONAL :: low, up
 REAL :: a, b
 INTEGER :: i, icount
 a = -HUGE(a)
 b = HUGE(b)
 IF (PRESENT(low)) a = low
 IF (PRESENT(up)) b = up
 average = 0.0
 icount = 0
 DO i = 1, SIZE(x)
 IF (x(i) >= a .AND. x(i) <= b) THEN
 average = average + x(i)
 icount = icount + 1
 END IF
 END DO
 average = average/icount
END FUNCTION average
```

Counting an average of a set of real numbers – optionally numbers outside [low,up] can be omitted from the average. The function can be called with either 1, 2 or 3 arguments, but the set of numbers has to be provided.

## Towards Fortran 2008

- The Fortran 2008 standard features e.g. the following capabilities on top of Fortran 2003
  - Submodules
  - Coarray Fortran** – a parallel execution model
    - Discussed tomorrow
  - The DO CONCURRENT construct
  - The CONTIGUOUS attribute
  - The BLOCK construct

## Command line input

- In many cases, it is convenient to give parameters for the program directly during program launch
  - Instead of using a parser, reading from an input file etc.
- Fortran 2003 provides a way for this
  - COMMAND\_ARGUMENT\_COUNT() : compute the number of user-provided arguments
  - GET\_COMMAND\_ARGUMENT(integer i, character arg(i)) extract the argument from position i
  - You will need internal I/O to convert e.g. integer-valued arguments into values of integer variables

## Towards Fortran 2008

"Coarrays were designed to answer the question: 'What is the smallest change required to convert Fortran into a robust and efficient parallel language?'"

The answer: a simple syntactic extension.

It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules."

John Reid,  
ISO Fortran Convener

# Towards Fortran 2008



See <http://fortranwiki.org/fortran/show/Fortran+2008+status>

## Fortran 2008 status

[Home Page](#) | [All Pages](#) | [Recently Revised](#) | [Authors](#) | [Feeds](#) | [Export](#) |

Compiler Support for the Fortran 2008 Standard

| Fortran 2008 features     | Absoft  | Cray | g95 | GNU     | HP | IBM | Intel   | NAG | Oracle | PathScale | PGI |
|---------------------------|---------|------|-----|---------|----|-----|---------|-----|--------|-----------|-----|
| Compiler Version Number   |         |      |     | 4.5     |    |     | 12.0    | 5.2 |        | 4.0       |     |
| Submodules                | N       | Y    | ?   | N       | N  | N   | N       | N   | N      | N         | N   |
| Coinarrays                | N       | Y    | P   | P (200) | N  | N   | Y       | N   | N      | N         | N   |
| Performance enhancements  |         |      |     |         |    |     |         |     |        |           |     |
| do concurrent             | N       | N    | ?   | P       | N  | N   | Y       | N   | N      | N         | N   |
| Contiguous attribute      | N       | Y    | ?   | Y       | N  | N   | Y       | N   | N      | N         | N   |
| Simply contiguous arrays  | N       | Y    | ?   | Y       | N  | N   | Y       | N   | N      | N         | N   |
| Data Declaration          |         |      |     |         |    |     |         |     |        |           |     |
| Maximum rank              | N       | N    | ?   | N       | N  | Y   | Y       | N   | N      | N         | N   |
| Long integers             | Y (100) | Y    | ?   | Y       | N  | Y   | Y (100) | Y   | Y      | Y         | Y   |
| Allocatable components of |         |      |     |         |    |     |         |     |        |           |     |





# Parallel Programming with Fortran Coarrays

Delivered at PRACE Advanced Training Centre,  
CSC – IT Center for Science Ltd, Finland,  
September 13, 2012

David Henty, Alan Simpson (EPCC)  
Harvey Richardson, Bill Long (Cray)



## Tutorial Overview

- The Fortran Programming Model in context
- Basic coarray features
- Practical Session 1
- Further coarray features
- Practical Session 2
- Advanced coarray features
- Practical Session 3
- Experiences with coarrays



2

## Programming models for HPC

- The challenge is to efficiently map a problem to the architecture we have
  - Take advantage of all computational resources
  - Manage distributed memories etc.
  - Optimal use of any communication networks
- The HPC industry has long experience in parallel programming
  - Vector, threading, data-parallel, message-passing etc.
- We would like to have models or combinations that are
  - efficient
  - safe
  - easy to learn and use



5

## Why consider new programming models?

- Next-generation architectures bring new challenges:
  - Very large numbers of processors with many cores
  - Complex memory hierarchy
  - even today (2011) we are at 500k cores
- Parallel programming is hard, need to make this simpler
- Some of the models we currently use are
  - bolt-ons to existing languages as APIs or directives
  - Hard to program for underlying architecture
  - unable to scale due to overheads
- So, is there an alternative to the models prevalent today?
  - Most popular are OpenMP and MPI ...



6

## The Fortran Programming Model in context



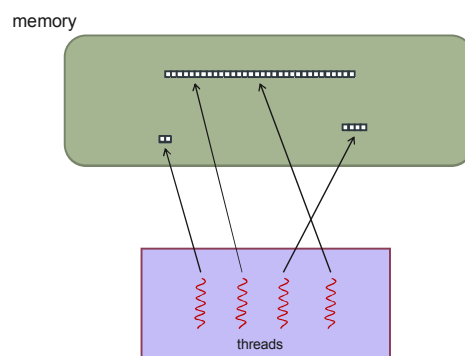
## Motivation

- Fortran now supports parallelism as a full first-class feature of the language
- Changes are minimal
- Performance is maintained
- Flexibility in expressing communication patterns



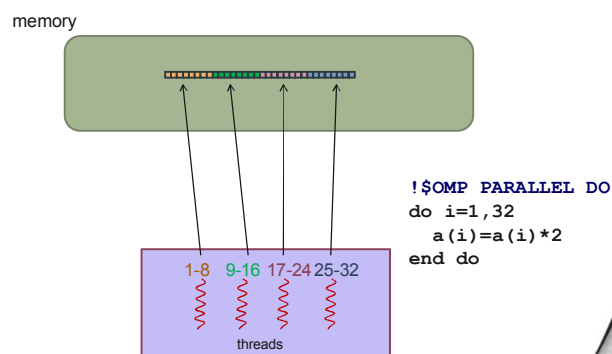
4

## Shared-memory directives and OpenMP



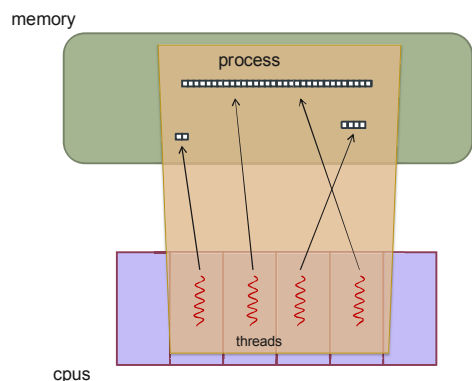
7

## OpenMP: work distribution

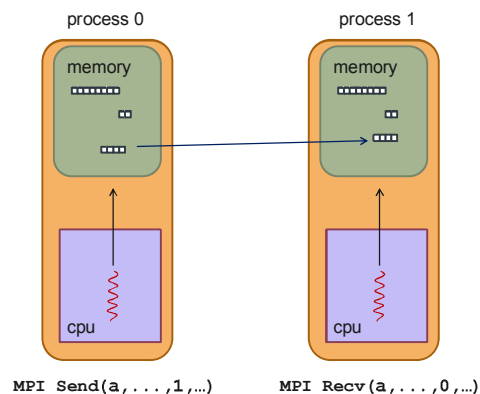


8

## OpenMP implementation



## MPI



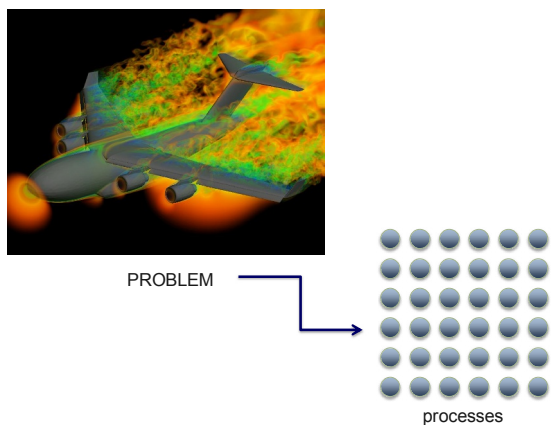
## Shared Memory Directives

- Multiple threads share global memory
- Most common variant: OpenMP
- Program loop iterations distributed to threads, more recent task features
  - Each thread has a means to refer to private objects within a parallel context
- Terminology
  - Thread, thread team
- Implementation
  - Threads map to user threads running on one SMP node
  - Extensions to distributed memory not so successful
- OpenMP is a good model to use within a node

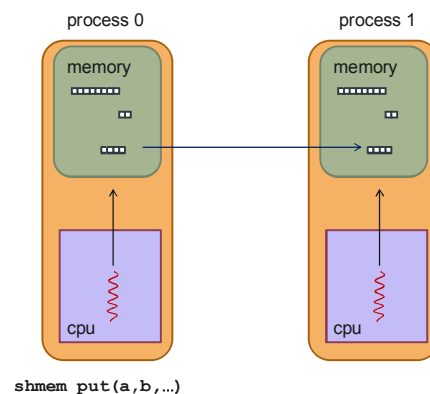
## Message Passing

- Participating processes communicate using a message-passing API
- Remote data can only be communicated (sent or received) via the API
- MPI (the Message Passing Interface) is the standard
- Implementation:
  - MPI processes map to processes within one SMP node or across multiple networked nodes
- API provides process numbering, point-to-point and collective messaging operations
- Mostly used in two-sided way, each endpoint coordinates in sending and receiving

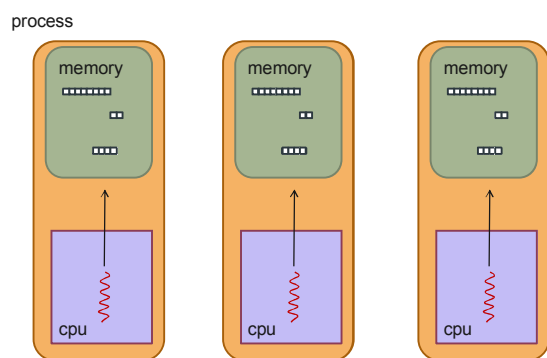
## Cooperating Processes Models



## SHMEM



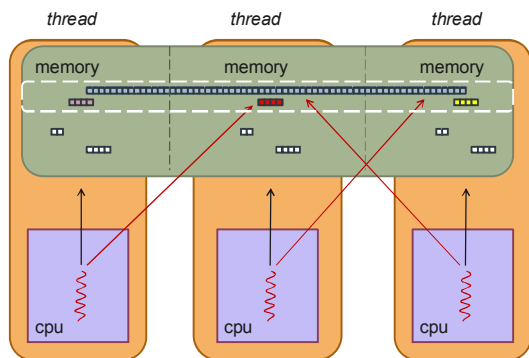
## Message Passing, MPI



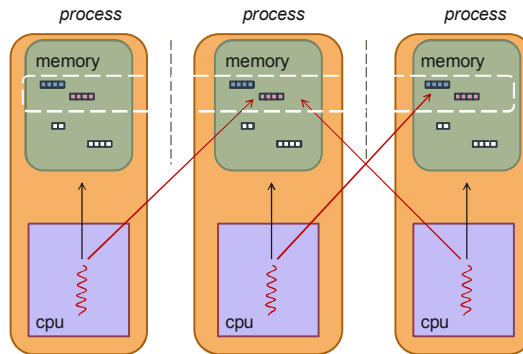
## SHMEM

- Participating processes communicate using an API
- Fundamental operations are based on one-sided PUT and GET
- Need to use symmetric memory locations
- Remote side of communication does not participate
- Can test for completion
- Barriers and collectives
- Popular on Cray and SGI hardware, also Blue Gene version
- To make sense needs hardware support for low-latency RDMA-type operations

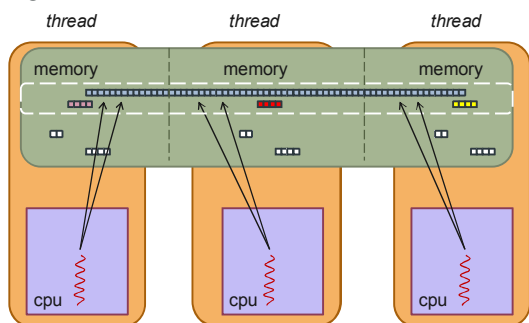
## UPC



## Fortran coarray model

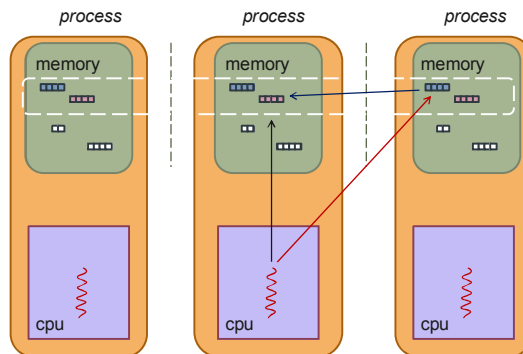


## UPC



```
upc_forall (i=0;i<32;i++;affinity) {
 a[i]=a[i]*2;
}
```

## Fortran coarray model



a = b[3]

## UPC

- Extension to ISO C99
- Participating "threads"
- New *shared* data structures
  - shared pointers to distributed data (block or cyclic)
  - pointers to shared data local to a thread
  - Synchronization
- Language constructs to divide up work on shared data
  - upc\_forall() to distribute iterations of for() loop
- Extensions for collectives
- Both commercial and open source compilers available
  - Cray, HP, IBM
  - Berkeley UPC (from LBL), GCC UPC

## Fortran coarrays

- Remote access is a full feature of the language:
  - Type checking
  - Opportunity to optimize communication
- No penalty for local memory access
- Single-sided programming model more natural for some algorithms
  - and a good match for modern networks with RDMA

## Fortran 2008 coarray model

- Example of a Partitioned Global Address Space (PGAS) model
- Set of participating processes like MPI
- Participating processes have access to **local memory** via standard program mechanisms
- Access to **remote memory** is directly supported by the language

## Fortran coarrays

Basic Features

## Coarray Fortran

"Coarrays were designed to answer the question:

*'What is the smallest change required to convert Fortran into a robust and efficient parallel language?'*

The answer: a simple syntactic extension.

It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules."

John Reid,  
ISO Fortran Convener

## Some History

- Introduced in current form by Numrich and Reid in 1998 as a simple extension to Fortran 95 for parallel processing
- Many years of experience, mainly on Cray hardware
- A set of core features are now part of the Fortran standard ISO/IEC 1539-1:2010
- Additional features are expected to be published in a Technical Specification in due course.

## How Does It Work?

- SPMD - Single Program, Multiple Data
  - single program replicated a fixed number of times
- Each replication is called an **image**
- Images are executed asynchronously
  - execution path may differ from image to image
  - some situations cause images to synchronize
- Images access remote data using *coarrays*
- Normal rules of Fortran apply

## What are coarrays?

- Arrays or scalars that can be accessed remotely
  - images can access data objects on any other image
- Additional Fortran syntax for coarrays
  - Specifying a codimension declares a coarray

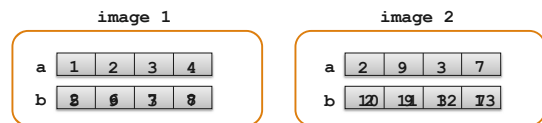
```
real, dimension(10), codimension[*] :: x
real :: x(10) [*]
```

- these are equivalent declarations of an array x of size 10 on each image
- x is now remotely accessible
- coarrays have the same size on each image!

## Accessing coarrays

```
integer :: a(4) [*], b(4) [*] !declare coarrays
b(:) = a(:) [n] ! copy
```

- integer arrays a and b declared to be size 4 on all images
- copy array a from remote image n into local array b
- () for local access [] for remote access
- e.g. for two images and n = 2:



## Synchronisation

- Be careful when updating coarrays:
  - If we get remote data was it valid?
  - Could another process send us data and overwrite something we have not yet used?
  - How do we know that data sent to us has arrived?
- Fortran provides synchronisation statements
- For example, barrier for synchronisation of all images:

```
sync all
```

- do not make assumptions about execution timing on images
  - unless executed after synchronisation
  - Note there is implicit synchronisation at program start

## Retrieving information about images

- Two intrinsics provide index of this image and number of images
  - this\_image() (image indexes start at 1)
  - num\_images()

```
real :: x[*]
if(this_image() == 1) then
 read *,x
 do image = 2,num_images()
 x[image] = x
 end do
end if
sync all
```

## Making remote references

- We used a loop over images

```
do image = 2,num_images()
 x[image] = x
end do
```

- Note that array indexing within the coindex is not allowed so we can not write

```
x[2:num_images()] = x ! illegal
```

## Data usage

- coarrays have the same size on every image
- Declarations:
  - round brackets () describe rank, shape and extent of local data
  - square brackets [] describe layout of images that hold local data
- Many HPC problems have physical quantities mapped to n-dimensional grids
- You need to implement your view of global data from the local coarrays as Fortran does not provide the global view
  - You can be flexible with the coindexing (see later)
  - You can use any access pattern you wish

## Example: maximum of array

```
real :: a(10)
real :: maximum[*]
call random_number(a)
maximum = maxval(a)
sync all
if (this_image() == 1) then
 do image = 2, num_images()
 maximum = max(maximum, maximum[image])
 end do
 do image = 2, num_images()
 maximum[image] = maximum
 end do
end if
sync all
```

implicit synchronisation

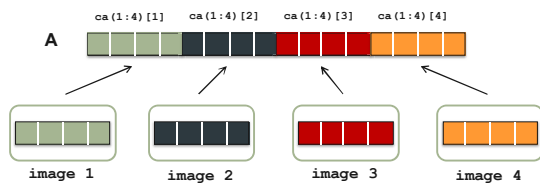
ensure all images set local maximum

ensure all images have copy of maximum value

## Data usage

- print out a 16 element "global" integer array A from 4 processors
  - 4 elements per processor = 4 coarrays on 4 images

```
integer :: ca(4)[*]
do image=1,num_images()
 print *,ca(:)[image]
end do
```



## Recap

We now know the basics of coarrays

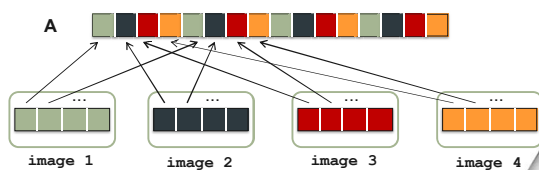
- declarations
- references with []
- this\_image() and num\_images()
- sync all

Now consider a full program example...

## 1D cyclic data access

- coarray declarations remain unchanged
  - but we use a cyclic access pattern

```
integer :: ca(4)[*]
do i=1,4
 do image=1,num_images()
 print *,ca(i)[image]
 end do
end do
```



## Example2: Calculate density of primes

```
program pdensity
 implicit none
 integer, parameter :: n=10000000, nimages=8
 integer start,end,i
 integer, dimension(nimages) :: nprimes[*]
 real density

 start = (this_image()-1) * n/num_images() + 1
 end = start + n/num_images() - 1

 nprimes(this_image())[1] = num_primes(start,end)

 sync all
```

## Synchronisation

- code execution on images is independent
  - programmer has to control execution using synchronisation
- synchronise before accessing coarrays
  - ensure content is not updated from remote images before you can use it
- synchronise after accessing coarrays
  - ensure new content is available to all images
- implicit synchronisation after variable declarations at first executable statement
  - guarantees coarrays exist on all images when your first program statement is executed
- We will revisit this topic later

## Example2: Calculate density of primes

```
...
if (this_image()==1) then
 nprimes(1)=sum(nprimes)
 density=real(nprimes(1))/n
 print *, "Calculating prime density on", &
 num_images(), "images"
 print *, nprimes(1), 'primes in', n, 'numbers'
 write(*, '(" density is ", 2Pf0.2, "%")') density
 write(*, '(" asymptotic theory gives ", &
 2Pf0.2, "%")') 1.0/(log(real(n))-1.0)
end if
```

## Example2: Calculate density of primes

```
Calculating prime density on 2 images
664580 primes in 10000000 numbers
density is 6.65%
asymptotic theory gives 6.61%
```

## Launching a coarray program

- The Fortran standard does not specify how a program is launched
- The number of images may be set at compile, link or run-time
- A compiler could optimize for a single image

Examples on Linux

- Cray XE  
aprun -n 16000 solver
- g95  
./solver --g95 -images=2

## Observations so far on coarrays

- Natural extension, easy to learn
- Makes parallel parts of program obvious (syntax)
- Part of Fortran language (type checking, etc)
- No mapping of data to buffers (or copying) or creation of complex types (as we might have with MPI)
- Compiler can optimize for communication
- More observations later...

## Exercise Session 1

- Look at the *Exercise Notes* document for full details
- Write, compile and run a “Hello World” program that prints out the value of the running image’s image index and the number of images
- Extend the simple Fortran code provided in order to perform operations on parts of a picture using coarrays

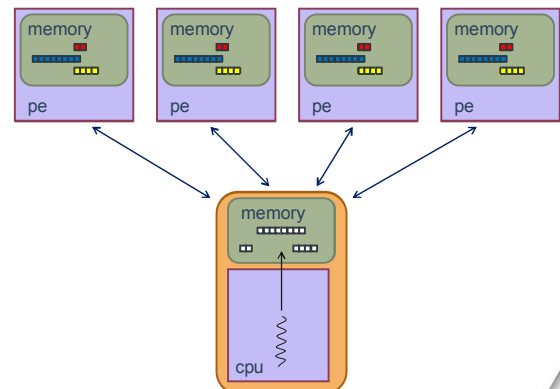
## Backup Slides

HPF model

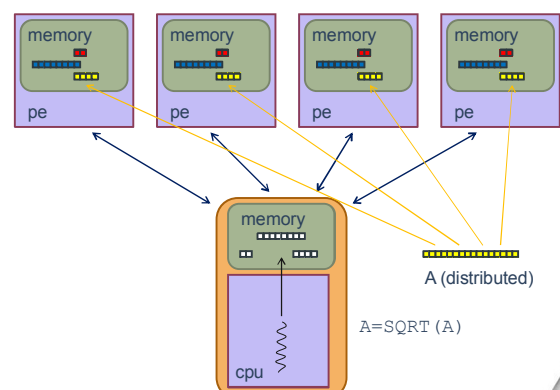
## High Performance Fortran (HPF)

- Data Parallel programming model
- Single thread of control
- Arrays can be distributed and operated on in parallel
- Loosely synchronous
- Parallelism mainly from Fortran 90 array syntax, FORALL and intrinsics.
- This model popular on SIMD hardware (AMT DAP, Connection Machines) but extended to clusters where control thread is replicated

## HPF



## HPF



## More Coarray Features

Parallel Programming with Fortran Coarrays  
Delivered at PRACE Advanced Training Centre,  
CSC – IT Center for Science Ltd, Finland,  
September 13, 2012

David Henty, Alan Simpson (EPCC)  
Harvey Richardson, Bill Long (Cray)



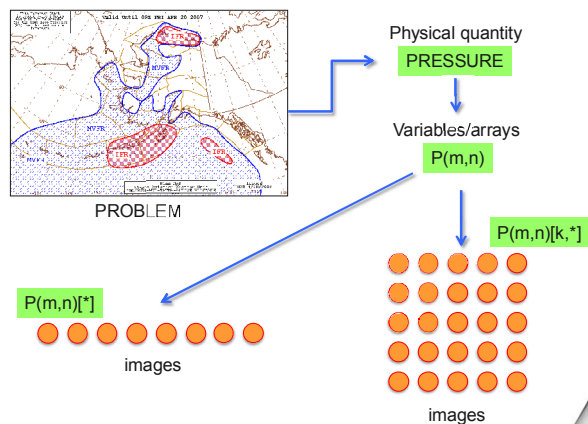
## Overview

- Multiple Dimensions and Codimensions
- Allocatable Coarrays and Components of Coarray Structures
- Coarrays and Procedures



2

## Mapping data to images

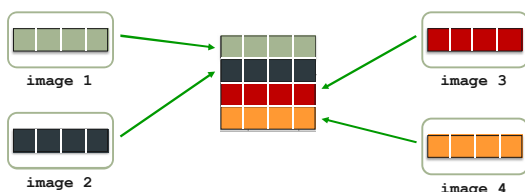


3

## 2D Data

- Coarray Fortran has a “bottom-up” approach to global data
  - assemble rather than distribute
  - unlike HPF (“top-down”) or UPC shared distributed data
- Can assemble a 2D data structure from 1D arrays

```
integer :: ca(4) [*]
```

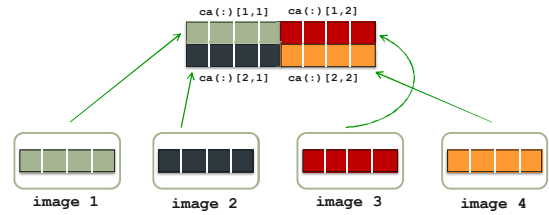


4

## 2D Data

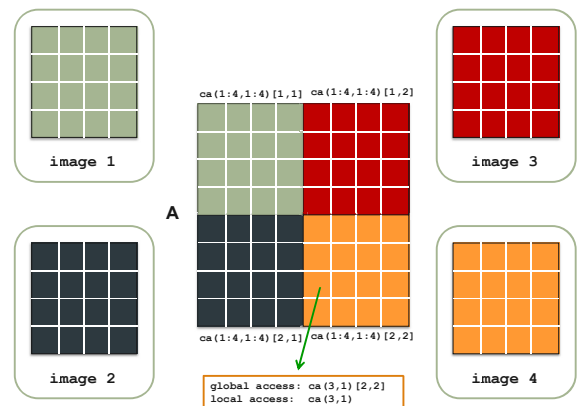
- However, images are not restricted to a 1D arrangement
- For example, we can arrange images in 2x2 grid
  - coarrays with 2 codimensions

```
integer :: ca(4) [2,*]
```



5

## 2D Local Array on 2D Grid



6

## Coarray Subscripts

- Fortran arrays defined by rank, bounds and shape
  - rank 2
  - lower bounds 1, 1; upper bounds 10, 4
  - shape [10, 4]
- Coarray Fortran adds corank, cobounds and coshape
  - corank 2
  - lower cobounds 1, 1; upper cobounds 3, m
  - coshape [3, m]
  - m would be `ceiling(num_images() / 3)`



7

## Multiple Codimensions

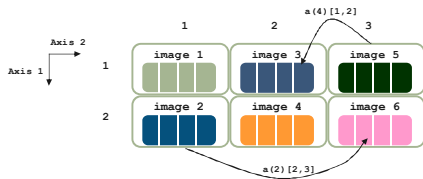
- Coarrays with multiple Codimensions:
  - `character :: a(4) [2, *] !2D grid of images`  
→ for 4 images, grid is 2x2; for 16 images, grid is 2x8
  - `real :: b(8,8,8) [10,5,*] !3D grid of images`  
→ 8x8x8 local array; with 150 images, grid is 10x5x3
  - `integer :: c(6,5) [0:9,0:*] !2D grid of images`  
→ lower cobounds [ 0, 0 ]; upper cobounds [ 9,n]  
→ useful if you want to interface with MPI or want C like coding
- Sum of rank and corank should not exceed 15
- Flexibility with cobounds
  - can set all but final upper cobound as required



8

## Codimensions: What They Mean

- Images are organised into a logical 2D, 3D, .... grid
  - for that coarray only
- A map so an image can find the coarray on any other image
  - access the coarray using its grid coordinates
- e.g. `character a(4) [2, *]` on 6 images
  - gives a 2 x 3 image grid
  - usual Fortran subscript order to determine image index



## Codimensions and Array-Element Order

- Storage order for multi-dimensional Fortran arrays
- Ordering of images in multi-dimensional cogrids

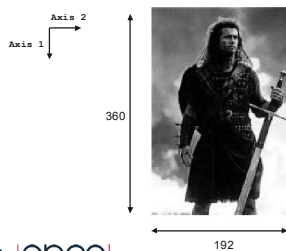
`real p(2,3,8)`

`real q(4) [2,3,*]`

| Location | Element  | Image | Elements      |
|----------|----------|-------|---------------|
| 1        | p(1,1,1) | 1     | q(1:4)[1,1,1] |
| 2        | p(2,1,1) | 2     | q(1:4)[2,1,1] |
| 3        | p(1,2,1) | 3     | q(1:4)[1,2,1] |
| 4        | p(2,2,1) | 4     | q(1:4)[2,2,1] |
| 5        | p(1,3,1) | 5     | q(1:4)[1,3,1] |
| 6        | p(2,3,1) | 6     | q(1:4)[2,3,1] |
| 7        | p(1,1,2) | 7     | q(1:4)[1,1,2] |
| 8        | p(2,1,2) | 8     | q(1:4)[2,1,2] |
| ...      | ...      | ...   | ...           |
| 48       | p(2,3,8) | 48    | q(1:4)[2,3,8] |
| ...      | ...      | ...   | ...           |

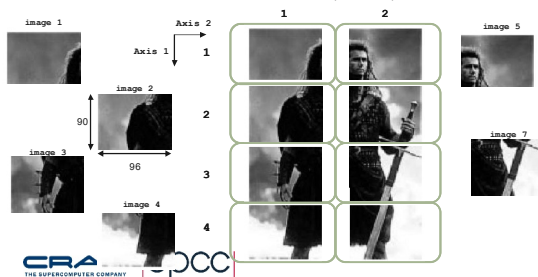
## Multi Codimensions: An Example

- Domain Decomposition
  - () gives local domain size
  - [] provides image grid and easy access to other images
- 2D domain decomposition of Braveheart
- Global data is 360 x 192
- Domain decomposition on 8 images with 4 x 2 grid
  - local array size: (360 / 4) x (192 / 2) = 90 x 96
  - declaration = `real :: localPic(90,96) [4,*]`



## Multi Codimensions: An Example

- Domain Decomposition
  - () gives local domain size
  - [] provides image grid and easy access to other images
- 2D domain decomposition of Braveheart
- Global data is 360 x 192
- Domain decomposition on 8 images with 4 x 2 grid
  - local array size: (360 / 4) x (192 / 2) = 90 x 96
  - declaration = `real :: localPic(90,96) [4,*]`



## this\_image() & image\_index()

`this_image()`

→ returns the image index, i.e., number between 1 and `num_images()`

`this_image(z)`

→ returns the rank-one integer array of cosubscripts for the calling image corresponding to the coarray `z`

→ `this_image(z, dim)` returns cosubscript of codimension `dim` of `z`

`image_index(z, sub)`

→ returns image index with cosubscripts `sub` for coarray `z`

→ `sub` is a rank-one integer array

## Example 1

PROGRAM CAF\_Intrinsics

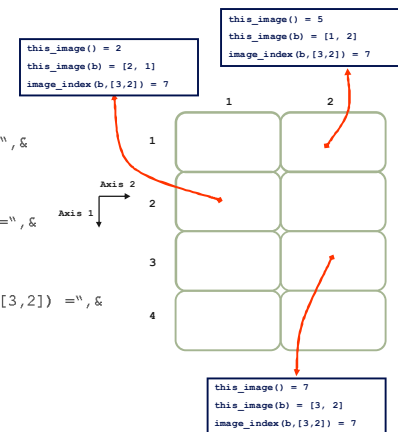
real :: b(90,96) [4,\*]

write(\*,\*) "this\_image() =", &  
this\_image()

write(\*,\*) "this\_image(b) =", &  
this\_image(b)

write(\*,\*) "image\_index(b,[3,2]) =", &  
image\_index(b,[3,2])

END PROGRAM CAF\_Intrinsics



## Example 2

PROGRAM CAF\_Intrinsics

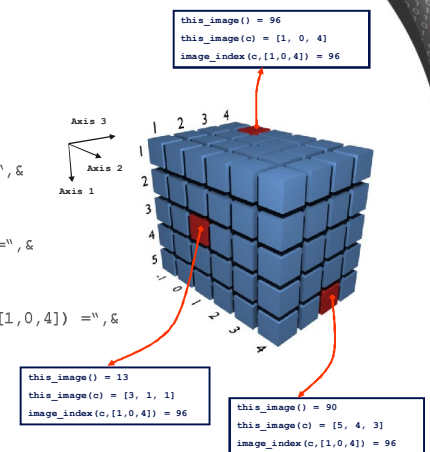
real :: c(4,4,4) [5,-1:4,\*]

write(\*,\*) "this\_image() =", &  
this\_image()

write(\*,\*) "this\_image(c) =", &  
this\_image(c)

write(\*,\*) "image\_index(c,[1,0,4]) =", &  
image\_index(c,[1,0,4])

END PROGRAM CAF\_Intrinsics



## Boundary Swapping

```
PROGRAM CAF_HaloSwap
integer, parameter :: nximages = 4, nyimages = 2
integer, parameter :: nxlocal = 90, nylocal = 96
real :: pic(0:nxlocal+1, 0:nylocal+1)[nximages,*] ! Declare coarray with halos
integer :: myimage(2) ! Array for my row & column coordinates
```

myimage = this\_image(pic) ! Find my row & column coordinates → Find cosubscripts

... ! Initialise pic on each image

sync all → Ensures pic initialised before accessed by other images

```
! Halo swap
if (myimage(1) > 1) &
pic(0,1:nylocal) = pic(nxlocal,1:nylocal) [myimage(1)-1, myimage(2)]
if (myimage(1) < nximages) &
pic(nxlocal+1,1:nylocal) = pic(1,1:nylocal) [myimage(1)+1, myimage(2)]

if (myimage(2) > 1) &
pic(1:nxlocal,0) = pic(1:nxlocal,nylocal) [myimage(1), myimage(2)-1]
if (myimage(2) < nyimages) &
pic(1:nxlocal,nylocal+1) = pic(1:nxlocal,1) [myimage(1), myimage(2)+1]
```

sync all → Ensures all images have got old values before pic is updated

... ! Update pic on each image

END PROGRAM CAF\_HaloSwap

## Allocatable Coarrays

- Can have allocatable Coarrays

```
real, allocatable :: x(:,:)[:], s[:,:]
n = num_images()
allocate(x(n)[*], s[4,*])
```
- Must specify cobounds in **allocate** statement
- The size and value of each bound and cobound must be same on all images.
  - **allocate(x(this\_image())[\*])** ! Not allowed
- Implicit synchronisation of all images...
  - ...after each **allocate** statement involving coarrays
  - ...before **deallocate** statements involving coarrays

## Differently Sized Coarray Components

- A coarray structure component can vary in size per image
- Declare a coarray of derived type with a component that is allocatable (or pointer)...

```
!Define data type with allocatable component
type diffSize
 real, allocatable :: data(:)
end type diffSize
!Declare coarray of type diffSize
type(diffSize) :: x[*]

! Allocate x\data to a different size on each image
allocate(x\data(this_image()))
```

## Pointer Coarray Structure Components

- We are allowed to have a coarray that contains components that are pointers
- Note that the pointers have to point to local data
- We can then access one of the pointers on a remote image to get at the data it points to
- This technique is useful when adding coarrays into an existing MPI code
  - We can insert coarray code deep in call tree without changing many subroutine argument lists
  - We don't need new coarray declarations
- Example follows...

## Pointer Coarray Structure Components...

- Existing non-coarray arrays u,v,w
- Create a type (coords) to hold pointers (x,y,z) that we use to point to x,y,z. We can use the vects coarray to access u, v, w.

```
subroutine calc(u,v,w)
real, intent(in), target, dimension(100) :: u,v,w
type coords
 real, pointer, dimension(:) :: x,y,z
end type coords
type(coords), save :: vects[*]
! ...
vects%x => u ; vects%y => v ; vects%z => w
sync all
firstx = vects[1]%x(1)
```

## Coarrays and Procedures

- An explicit interface is required if a dummy argument is a coarray
- Dummy argument associated with coarray, not a copy
  - avoids synchronisation on entry and return
- Other restrictions on passing coarrays are:
  - the actual argument should be contiguous
    - **a(:,2)** is OK, but **a(2,:)** is not contiguous
  - or the dummy argument should be assumed shape ... to avoid copying
- Function results cannot be coarrays

## Coarrays as Dummy Arguments

- As with standard Fortran arrays, the coarray dummy arguments in procedures can be:
  - Explicit shape: each dimension of a coarray declared with explicit value
  - Assumed shape: extents and bounds determined by actual array
  - Assumed size: only size determined from actual array
  - Allocatable: the size and shape can be determined at run-time

```
subroutine s(n, a, b, c, d)
integer :: n
real :: a(n) [n,*] ! explicit shape - permitted
real :: b(:,) [*] ! assumed shape - permitted
real :: c(n,*) [*] ! assumed size - permitted
real, allocatable :: d(:)[:,] ! allocatable - permitted
```

## Assumed Size Coarrays

- Allow the coshape to be remapped to corank 1

```
program cmax
real, codimension[8,*] :: a(100), amax

 a = [(i, i=1,100)] * this_image() / 100.0
 amax = maxval(a)
 sync all
 amax = AllReduce_max(amax)
 ...
contains
 real function AllReduce_max(r) result(rmax)
 real :: r[*]
 sync all
 rmax = r
 do i=1,num_images()
 rmax = max(rmax, r[i])
 end do
 sync all
end function AllReduce_max
```

## Coarrays Local to a Procedure

- Coarrays declared in procedures must have **save** attribute
  - unless they are dummy arguments or allocatable
  - save attribute: retains value between procedure calls
  - avoids synchronisation on entry and return
- Automatic coarrays are not permitted
  - Automatic array: local array whose size depends on dummy arguments
  - would require synchronisation for memory allocation and deallocation
  - would need to ensure coarrays have same size on all images

```
subroutine t(n)
integer :: n
real :: temp(n) [*] ! automatic - not permitted
integer, save :: x(4) [*] ! coarray with save attribute
integer :: y(4) [*] ! not saved - not permitted
```

## Summary

- Coarrays with multiple codimensions used to create a grid of images
  - `()` gives local domain information
  - `[]` gives an image grid with easy access to other images
- Can be used in various ways to assemble a multi-dimensional data set
- `this_image()` and `image_index()`
  - are intrinsic functions that give information about the images in an multi-codimension grid
- Flexibility from non-coarray allocatable and pointer components of coarray structures
- Coarrays can be allocatable, can be passed as arguments to procedures, and can be dummy arguments

# Advanced Features

Parallel Programming with Fortran Coarrays  
Delivered at PRACE Advanced Training Centre,  
CSC – IT Center for Science Ltd, Finland,  
September 13, 2012

David Henty, Alan Simpson (EPCC)  
Harvey Richardson, Bill Long (Cray)



## Advanced Features: Overview

- Execution segments and Synchronisation
- Non-global Synchronisation
- Critical Sections
- Visibility of changes to memory
- Other Ininsics
- Miscellaneous features
- Future developments



2

## More on Synchronisation

- We have to be careful with one-sided updates
  - If we read remote data, was it valid?
  - Could another process send us data and overwrite something we have not yet used?
  - How do we know when remote data has arrived?
- The standard introduces **execution segments** to deal with this: segments are bounded by image control statements

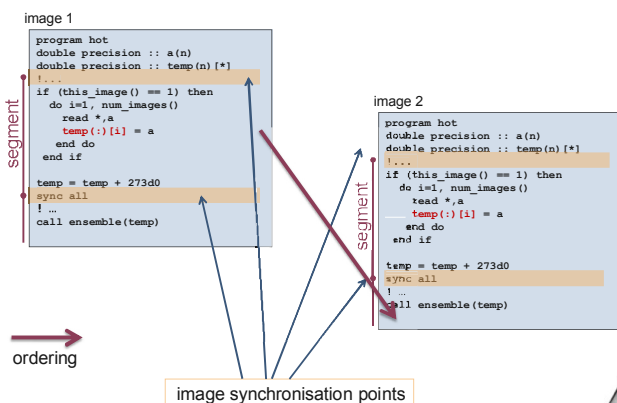
The standard can be summarized as follows:

- If a variable is defined in a segment, it must not be referenced, defined, or become undefined in another segment unless the segments are ordered – *John Reid*



3

## Execution Segments



4

## Synchronisation mistakes

- This code is wrong

```
subroutine allreduce_max_getput(v,vmax)
 double precision, intent(in) :: v[*]
 double precision, intent(out) :: vmax[*]
 integer i
 sync all

 vmax=v
 if (this_image()==1) then
 do i=2,num_images()
 vmax=max(vmax,v[i])
 end do
 do i=2,num_images()
 vmax[i]=vmax
 end do
 end if
 sync all
```



5

## Synchronisation mistakes

- It breaks the rules

```
subroutine allreduce_max_getput(v,vmax)
 double precision, intent(in) :: v[*]
 double precision, intent(out) :: vmax[*]
 integer i
 sync all

 vmax=v
 if (this_image()==1) then
 do i=2,num_images()
 vmax=max(vmax,v[i])
 end do
 do i=2,num_images()
 vmax[i]=vmax
 end do
 end if
 sync all
```



6

## Synchronisation mistakes

- This is ok

```
subroutine allreduce_max_getput(v,vmax)
 double precision, intent(in) :: v[*]
 double precision, intent(out) :: vmax[*]
 integer i
 sync all

 if (this_image()==1) then
 vmax=v
 do i=2,num_images()
 vmax=max(vmax,v[i])
 end do
 do i=2,num_images()
 vmax[i]=vmax
 end do
 end if
 sync all
```



7

## More about sync all

- Usually all images execute the same **sync all** statement
- But this is not a requirement...
  - Images execute different code with different **sync all** statements
  - All images execute the first **sync all** they come across and....
    - this may match an arbitrary **sync all** on another image
    - causing incorrect execution and/or deadlock
- Need to be careful with this 'feature'
  - Possible to write code which doesn't deadlock but gives wrong answers



8

## More about `sync all`

- e.g. Image practical: wrong answer

```
! Do halo swap, taking care at the upper and lower picture boundaries
if (myimage < numimage) then
 oldpic(1:nxlocal, nylocal+1) = oldpic(1:nxlocal, 1)[myimage+1]
 sync all -----> All images NOT executing this sync all
end if

! ... and the same for down halo
! Now update the local values of newpic
...

! Need to synchronise to ensure that all images have finished reading the
! oldpic halo values on this image before overwriting it with newpic
sync all -----> All images ARE executing this sync all
oldpic(1:nxlocal, 1:nylocal) = newpic(1:nxlocal, 1:nylocal)

! Need to synchronise to ensure that all images have finished updating
! their oldpic arrays before this image reads any halo data from them
sync all
```

## Other Synchronisation

- `lock` and `unlock` statements

- Control access to data defined or referenced by more than one image
  - as opposed to `critical` which controls access to lines of code
  - USE `iso_fortran_env` module and define coarray of `type(lock_type)`
- e.g. to lock data on image 2

```
type(lock_type) :: qLock[*]

lock(qLock[2])
!access data on image 2
unlock(qLock[2])
```

## More about `sync all`

- `sync images(imageList)`
  - Performs a synchronisation of the image executing `sync images` with each of the images specified in `imageList`
  - `imageList` can be an array or a scalar

```
if (myimage < numimage) then
 oldpic(1:nxlocal, nylocal+1) = oldpic(1:nxlocal, 1)[myimage+1]
end if
if (myimage > 1) then
 oldpic(1:nxlocal, 0) = oldpic(1:nxlocal, nylocal)[myimage-1]
end if

! Now perform local pairwise synchronisations
if (myimage == 1) then
 sync images(2)
else if (myimage == numimage) then
 sync images(numimage-1)
else
 sync images((/ myimage-1, myimage+1 /))
end if
```

## Other Intrinsic functions

- `lcobound(z)`
  - Returns lower cobounds of the coarray `z`
  - `lcobound(z,dim)` returns lower cobound for codimension `dim` of `z`
- `ucobound(z)`
  - Returns upper cobounds of the coarray `z`
  - `lcobound(z,dim)` returns upper cobound for codimension `dim` of `z`

- `real :: array(10) [4,0:*]` on 16 images
  - `lcobound(array)` returns [ 1, 0 ]
  - `ucobound(array)` returns [ 4, 3 ]

## Other Synchronisation

- Critical sections
  - Limit execution of a piece of code to one image at a time
  - e.g. calculating global sum on master image

```
integer :: a(100) [*]
integer :: globalSum[*] = 0, localSum
... ! Initialise a on each image

localSum = SUM(a) !Find localSum of a on each image

critical
 globalSum[1] = globalSum[1] + localSum
end critical
```

## More on Cosubscripts

- `integer :: a[*]` on 8 images
  - cosubscript `a[9]` is not valid
- `real :: b(10) [3,*]` on 8 images
  - `ucobound(b)` returns [ 3, 3 ]
  - cosubscript `b[2,3]` is valid (corresponds to image 8)...
  - ...but cosubscript `b[3,3]` is invalid (image 9)
- Programmer needs to make sure that cosubscripts are valid
  - `image_index` returns 0 for invalid cosubscripts

## Other Synchronisation

- `sync memory`
  - Coarray data held in caches/registers made visible to all images
    - requires some other synchronisation to be useful
    - unlikely to be used in most coarray codes
- Example usage: Mixing MPI and coarrays
  - loop: coarray operations
  - sync memory
  - call `MPI_Allreduce(...)`
- `sync memory` implied for `sync all` and `sync images`

## Assumed Size Coarrays

- Codimensions can be remapped to corank greater than 1
  - useful for determining optimal extents at runtime

```
program 2d
real, codimension[*] :: picture(100,100)
integer :: numimage, numimagex, numimagey
numimage = num_images()

call get_best_2d_decomposition(numimage, &
 numimagex, numimagey)
! Assume this ensures numimage=numimagex*numimagey

call dothework(picture, numimagex, numimagey)
...
contains
 subroutine dothework(array, m, n)
 real, codimension[m,*] :: array(100,100)
 ...
 end subroutine dothework
```

## I/O

- Each image has its own set of input/output units
- units are independent on each image
- Default input unit is preconnected on image 1 only
  - `read *,... ,read(*,...) ...`
- Default output unit is available on all images
  - `print *,... ,write(*,...) ...`
  - It is expected that the implementation will merge records from each image into one stream

## Program Termination

- STOP or END PROGRAM statements initiate *normal termination* which includes a synchronisation step
- An image's data is still available after it has initiated normal termination
- Other images can test for this using STAT= specifier to synchronisation calls or allocate/deallocate
  - test for STAT\_STOPPED\_IMAGE (defined in ISO\_FORTRAN\_ENV module)
- The ERROR STOP statement initiates error termination and it is expected all images will be terminated.

## Coarray Technical Specification

- Additional coarray features may be described in a Technical Specification (TS)
- Developed as part of the official ISO standards process
- Work in progress and the areas of discussion are:
  - image teams
  - collective intrinsics for coarrays
  - atomics

## TS: Teams

- Often useful to consider subsets of processes
  - e.g. MPI communicators
- Subsets not currently supported in Fortran, e.g.
  - `sync all`: all images
  - `sync images`: pairwise mutual synchronisation
- Extension involves TEAMS of images
  - user creates teams
  - specified as an argument to various functions

## TS: Teams...

- To define a set of images as a **team**  

```
call form_team(oddtteam, [(i,i=1,n,2)])
```
- To synchronise the team  

```
sync team(oddtteam)
```
- To determine images that constitute a team:  

```
oddtimages=team_images(oddtteam)
```

## TS: Collectives

- Collective operations a key part of real codes
  - broadcast
  - global sum
  - ...
- Supported in other parallel models
  - OpenMP reductions
  - **MPI\_Allreduce**
- Not currently supported for coarrays
  - efficient implementation by hand is difficult
  - calling external MPI routines rather ugly

## TS: Collective intrinsic subroutines

- Collectives, with in/out arguments, invoked by same statement on all images (or team of images)
- Routines
  - CO\_BCAST
  - CO\_SUM and other reduction operations
  - basically reproduce the MPI functionality
- Arguments include SOURCE, RESULT, TEAM
- Still discussion on need for implicit synchronisation and argument types (for example non-coarray arguments)

## TS: Atomic operations

- Critical or lock synchronisation sometimes overkill
  - `counter[1] = counter[1] + 1`
- Simple atomic operations can be optimised
  - e.g. OpenMP **atomic**  

```
!$OMP atomic
sharedcounter = sharedcounter + 1
```
- New variable types and operations for coarrays

## TS: Atomic variables

- Fortran already includes some atomic support (define,ref)
- TS expands on this to supports atomic compare and swap, fetch and add, ...

```
integer (atomic_int_kind) :: counter[*]

call atomic_define(counter[1],0)
call atomic_add(counter[1],1)
call atomic_ref(countval,counter[1])
```

# Experiences with Coarrays

Parallel Programming with Fortran Coarrays  
Delivered at PRACE Advanced Training Centre,  
CSC – IT Center for Science Ltd, Finland,  
September 13, 2012

David Henty, Alan Simpson (EPCC)  
Harvey Richardson, Bill Long (Cray)



## Overview

- Implementations
- Performance considerations
- Where to use the coarray model
- Coarray benchmark suite
- Examples of coarrays in practice
- References
- Wrapup



2

## Implementations we have used

- Cray X1/X2
  - Hardware supports communication by direct load/store
  - Very efficient with low overhead
- Cray XT
  - PGAS (UPC, CAF) layered on GASNet/portals (so messaging)
  - Not that efficient
- Cray XE
  - PGAS layered on DMAPP portable layer over Gemini network hardware
  - Intermediate between XT and X1/2
- Intel Composer XE 2011
  - SMP and message-passing runtimes



5

## Implementations we have used...

- g95 on shared-memory
  - Using cloned process images on Linux
  - This is not being actively developed



6

## Implementation Status

- History of coarrays dates back to Cray implementations
- Expect support from vendors as part of Fortran 2008
- G95 had multi-image support in 2010
  - has not been updated for some time
- gfortran
  - Introduced single-image support at version 4.6
- Intel: multi-process coarray support in Intel Composer XE 2011 (based on Fortran 2008 draft)
- Runtimes are SMP, GASNet and compiler/vendor runtimes
  - GASNet has support for multiple environments (IB, Myrinet, MPI, UDP and Cray/IBM systems) so could be an option for new implementations



3

## Intel XE on Ubuntu VM

```
harvey@u10-vbox:~/fortran$ ifort -coarray -03 -o pdensity pdensity.f90
harvey@u10-vbox:~/fortran$ export FOR_COARRAY_NUM_IMAGES=1
harvey@u10-vbox:~/fortran$./pdensity
Calculating prime density on 1 images
539778 primes in 8000000 numbers
density is 6.75%
asymptotic theory gives 6.71%
Done in 15.61 seconds
harvey@u10-vbox:~/fortran$ export FOR_COARRAY_NUM_IMAGES=2
harvey@u10-vbox:~/fortran$./pdensity
Calculating prime density on 2 images
539778 primes in 8000000 numbers
density is 6.75%
asymptotic theory gives 6.71%
Done in 10.31 seconds
harvey@u10-vbox:~/fortran$ _
```



7

## Implementation Status (Cray)

- Cray has supported coarrays and UPC on various architectures over the last decade (from T3E)
- Full PGAS support on the Cray XT/XE
  - Cray Compiling Environment 7.0 – Dec 2008
  - Current release is Cray Compiler Environment 7.4
  - Full Fortran 2008 coarray support
  - Full Fortran 2003 with some Fortran 2008 features
- Fully integrated with the Cray software stack
  - Same compiler drivers, job launch tools, libraries
  - Integrated with Craypat – Cray performance tools
  - Can mix MPI and coarrays



4

## When to use coarrays

- Two obvious contexts
  - Complete application using coarrays
  - Mixed with MPI
- As an incremental addition to a (potentially large) serial code
- As an incremental addition to an MPI code (allowing reuse of most of the existing code)
- Use coarrays for some of the communication
  - opportunity to express communication much more simply
  - opportunity to overlap communication
- For subset synchronisation
- Work-sharing schemes



8

## Adding coarrays to existing applications

- Constrain use of coarrays to part of application
  - Move relevant data into coarrays
  - Implement parallel part with coarray syntax
  - Move data back to original structures
- Use coarray structures to contain pointers to existing data
- Place relevant arrays in global scope (modules)
  - avoids multiple declarations
- Declare existing arrays as coarrays at top level and through the complete call tree (some effort but only requires changes to declarations)

## Performance Considerations

- What is the latency?
- Do you need to avoid strided transfers?
- Is the compiler optimising the communication for target architecture?
  - Is it using blocking communication within a segment when it does not need to?
  - Is it optimising strided communication?
  - Can it pattern-match loops to single communication primitives or collectives?

## Performance: Communication patterns

- Try to avoid creating traffic jams on the network, such as all images storing to a single image.
- The following examples show two ways to implement an `ALLReduce()` function using coarrays

## ALLReduce (everyone gets)

- All images get data from others simultaneously

```
function allreduce_max_allget(v) result(vmax)
 double precision :: vmax, v[*]
 integer i

 sync all

 vmax=v
 do i=1,num_images()
 vmax=max(vmax,v[i])
 end do
```

## ALLReduce (everyone gets, optimized)

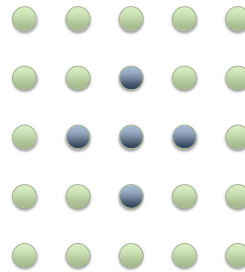
- All images get data from others simultaneously but this is optimized so communication is more balanced

```
!...
sync all
vmax=v
do i=this_image()+1,num_images()
 vmax=max(vmax,v[i])
end do
do i=1,this_image()-1
 vmax=max(vmax,v[i])
end do
```

- Have seen this much faster

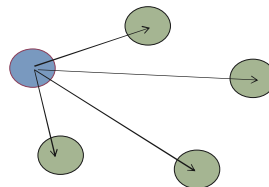
## Synchronization

- For some algorithms (finite-difference etc.) don't use `sync all` but pairwise synchronization using `sync images(image)`



## Synchronization (one to many)

- Often one image will be communicating with a set of images
- In general not a good thing to do but assume we are...

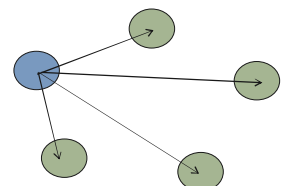


- Tempting to use `sync all`

## Synchronisation (one to many)

- If this is all images then could do

```
if (this_image() == 1) then
 sync images(*)
else
 sync images(1)
end if
```

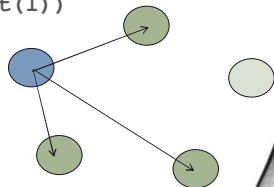


- Note that `sync all` is likely to be fast so is an alternative

## Synchronisation (one to many)

- For a subset use this

```
if (this_image() == image_list(1)) then
 sync images(image_list)
else
 sync images(image_list(1))
end if
```



- instead of `sync images(image_list)` for all of them which is likely to be slower

## Collective Operations

- If you need scalability to a large number of images you may need to temporarily work around current lack of collectives
  - Use MPI for the collectives if MPI+coarrays is supported
  - Implement your own but this might be hard
    - For reductions of scalars a tree will be the best to try
    - For reductions of more data you would have to experiment and this may depend on the topology
- Coarrays can be good for collective operations where
  - there is an unusual communication pattern that does not match what MPI collectives provide
  - there is opportunity to overlap communication with computation

## Tools: debugging and profiling

- Tool support should improve once coarray takeup increases
- Cray Craypat tool supports coarrays
- Totalview works with coarray programs on Cray systems
- Allinea DDT
  - support for coarrays and UPC for a number of compilers is in public beta and will be in DDT 3.1
- Scalasca
  - Currently investigating how PGAS support can be incorporated.

## Debugging Synchronisation problems

- One-sided model is tricky because subtle synchronisation errors change data
- TRY TO GET IT RIGHT FIRST TIME
  - look carefully at the remote operations in the code
  - Think about synchronisation of segments
  - especially look for early arriving communications trashing your data at the start of loops (this one is easy to miss)
- One way to test is to put `sleep()` calls in the code
  - Delay one or more images
  - Delay master image, or other images for some patterns

## Coarray Benchmark Suite

- Developed by David Henty at EPCC
- Aims to test fundamental features of a coarray implementation
- We don't have an API to test (cf. IMB for MPI)
- We can test basic language syntax for communication of data and synchronization
- Need to choose communication pattern and data access
- There is some scope for a given communication pattern:
  - array syntax, loops over array elements
  - inline code or use subroutines
  - Choices can reveal compiler capabilities

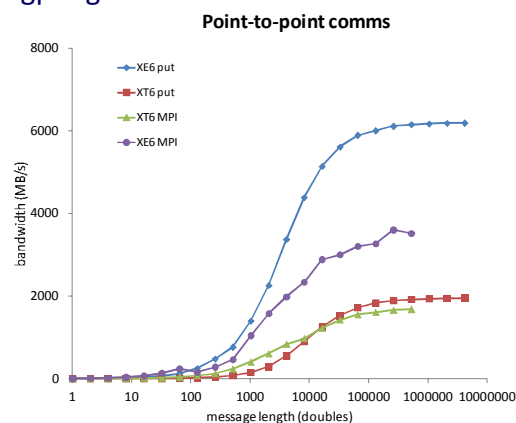
## Initial Benchmark

- Single contiguous point-to-point read and write
- Multiple contiguous point-to-point read and write
- Strided point-to-point read and write
- All basic synchronization operations
- Various representative communication patterns
- Halo-swap in multi-dimensional regular domain decomposition
- All communications include synchronisation cost
  - use double precision (8-byte) values as the basic type
  - use Fortran array syntax
- Synchronisations overhead measured separately as  $\text{overhead} = (\text{delay} + \text{sync}) - \text{delay}$

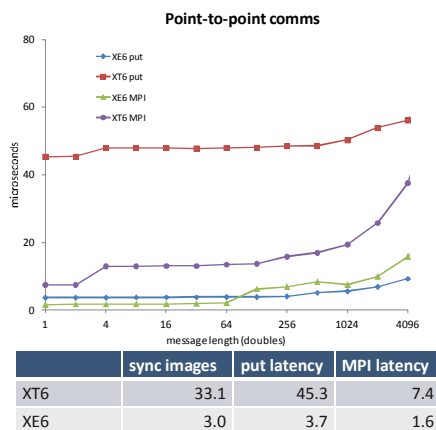
## Platforms

- Limited compiler support at present
  - results presented from Cray systems
- Cray Compiler Environment (CCE) 7.4.1
  - Cray XT6  
MPP with dual 12-core Opteron Magny-Cours nodes and Cray Seastar2+ torus interconnect.  
Coarrays implemented using GASNET.
  - Cray XE6  
Same nodes as XT6 but with Gemini torus interconnect which support RDMA

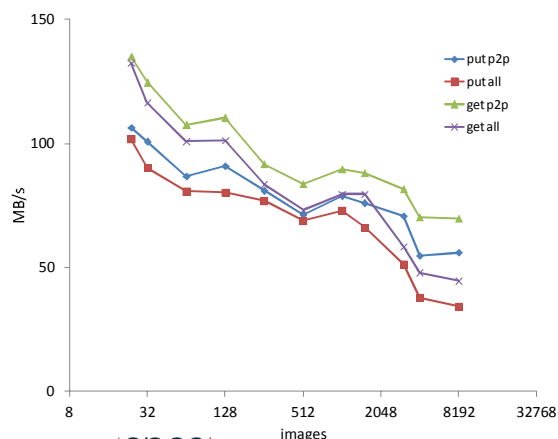
## Pingpong



## Pingpong (small message regime)

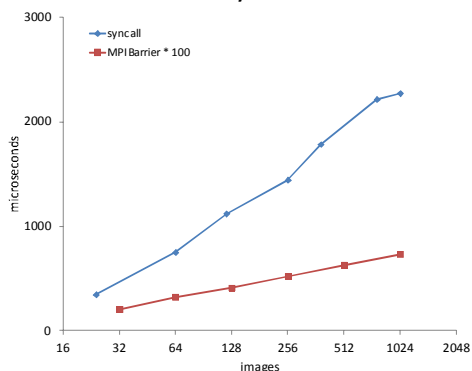


## 3D Halo Swap on XE6 (weak scaling $V=50^3$ )



## Global Synchronisation

### XT6 Synchronisation



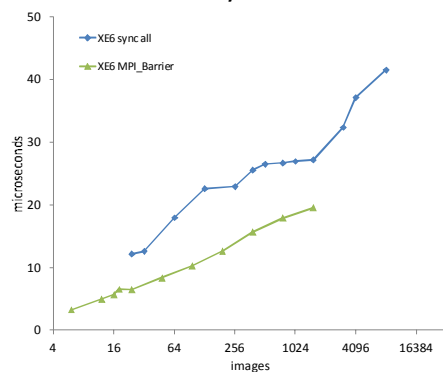
- XT coarray implementation not keeping up with MPI

## Examples of coarrays in practice

- Puzzles
- Distributed Remote Gather
- HIMENO Halo-Swap
- Gyrokinetic Fusion Code

## Global Synchronisation

### XE6 Synchronisation



- Much faster than the previous XT results

## Solving Sudoku Puzzles

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 3 |   | 7 |   |   |   |   |
| 6 |   |   | 1 | 9 | 5 |   |   |
|   | 9 | 8 |   |   |   | 6 |   |
| 8 |   |   | 6 |   |   |   | 3 |
| 4 |   |   | 8 |   | 3 |   | 1 |
| 7 |   |   | 2 |   |   |   | 6 |
|   | 6 |   |   |   | 2 | 8 |   |
|   |   |   | 4 | 1 | 9 |   | 5 |
|   |   |   | 8 |   |   | 7 | 9 |

## Global Synchronisation

- Also measured **sync images**
  - various point-to-point patterns
- Observed that **sync images** is usually faster than **sync all** on more than 512 images

## Going Parallel

- Started with serial code
- Changed to read in all 125,000 puzzles at start
- Choose work-sharing strategy
  - One image (1) holds a queue of puzzles to solve
  - Each image picks work from the queue and writes result back to queue
- Arbitrarily decide to parcel work as  
 $\text{blocksize} = \text{npuzzles} / (8 * \text{num\_images}())$

## Data Structures

```
use, intrinsic iso_fortran_env

type puzzle
 integer :: input(9,9)
 integer :: solution(9,9)
end type puzzle

type queue
 type (lock_type) :: lock
 integer :: next_available = 1
 type(puzzle), allocatable :: puzzles(:)
end type queue

type(queue), save :: workqueue[*]
type(puzzle) :: local_puzzle
integer, save :: npuzzles[*], blocksize[*]
```

## Solve the puzzles and write back

```
! Solve those puzzles

do i = istart,iend

 local_puzzle%input = &
 & workqueue[1]%puzzles(i)%input

 call sudoku_solve &
 & (local_puzzle%input, local_puzzle%solution)

 workqueue[1]%puzzles(i)%solution = &
 & local_puzzle%solution

end do
```

## Input

```
if (this_image() == 1) then
 ! After file Setup.
 inquire (unit=inunit, size=nbytes)
 nrecords = nbytes/10
 npuzzles = nrecords/9
 blocksize = npuzzles / (num_images()*8)

 write (*,*) "Found ", npuzzles, " puzzles."
 allocate (workqueue%puzzles(npuzzles))
 do i = 1, npuzzles
 call read_puzzles(&
 & workqueue%puzzles(i)%input, inunit, &
 & error)
 end do

 close(inunit)
```

## Output the solutions

```
! Need to synchronize puzzle output updates
sync all

if (this_image() == 1) then

 open (outunit, file=outfile, iostat=error)

 do i = 1, npuzzles
 call write_puzzle &
 & (workqueue%puzzles(i)%input, &
 & workqueue%puzzles(i)%solution, outunit, error)
 end do
```

## Core program structure

```
! After coarray data loaded
sync all

blocksize = blocksize[1]
npuzzles = npuzzles[1]

done = .false.
workloop: do

 ! Acquire lock and claim work

 ! Solve our puzzles

end do workloop
```

## More on the Locking

- We protected access to the queue state by lock and unlock
- During this time no other image can acquire the lock
  - We need to have discipline to only access data within the window when we have the lock
  - There is no connection with the lock variable and the other elements of the queue structure
- The unlock is acting like sync memory
  - If one image executes an unlock...
  - Another image getting the lock is ordered after the first image

## Acquire lock and claim work

```
! Reserve the next block of puzzles

lock (workqueue[1]%lock)
next = workqueue[1]%next_available
if (next <= npuzzles) then
 istart = next
 iend = min(npuzzles, next+blocksize-1)
 workqueue[1]%next_available = iend+1
else
 done = .true.
end if

unlock (workqueue[1]%lock)
if (done) exit workloop
```

## Summary and Commentary

- We implemented solving the puzzles using a work-sharing scheme with coarrays
- Scalability limited by serial work done by image 1
- I/O
  - Parallel I/O (deferred to TS) with multiple images running distributed work queues.
  - Defer the character-integer format conversion to the solver, which is executed in parallel.
- Lock contention
  - Could use distributed work queues, each with its own lock.

## Distributed remote gather

- The problem is how to implement the following gather loop on a distributed memory system

```
REAL :: table(n), buffer(nelts)
INTEGER :: index(nelts) ! nelts << n
...
DO i = 1, nelts
 buffer(i) = table(index(i))
ENDDO
```

- The array **table** is distributed across the processors, while **index** and **buffer** are replicated
- Synthetic code, but simulates “irregular” communication access patterns

## Remote gather: MPI implementation

- MPI rank 0 controls the index and receives the values from the other ranks

```
IF (mype.eq.0) THEN
 isum=0
 ! PEO gathers indices to send out to individual PEs
 DO i=1,nelts
 pe=(index(i)-1)/nloc
 isum(pe)=isum(pe)+1
 who(isum(pe),pe)=index(i)
 ENDDO
 ! send out count and indices to PEs
 DO i=1,npe-1
 CALL MPI_SEND(isum(i),1,MPI_INTEGER,i,10,
 IF(isum(i).gt.0) THEN
 CALL MPI_SEND(who(1,i),isum(i),...
 ENDDO
 ENDDO
 ! now wait to receive values and scatter them.
 DO i=1,isum(0)
 offset=mod(who(i,0)-1,nloc)+1
 buff(i,0)=mpi_table(offset)
 ENDDO
 DO i=1,npe-1
 IF(isum(i).gt.0) THEN
 CALL MPI_RECV(buff(1,i),isum(i),...
 ENDDO
ELSE ! If my_rank.ne.0
 ! Each PE gets the list and sends the values to PEO
 CALL MPI_RECV(my_sum,1,MPI_INTEGER,...
 IF(my_sum.gt.0) THEN
 CALL MPI_RECV(index,my_sum,MPI_INTEGER,...
 DO i=1,my_sum
 offset=mod(index(i)-1,nloc)+1
 mpi_buffer(i)=mpi_table(offset)
 ENDDO
 CALL MPI_SEND(mpi_buffer,my_sum,...
 ENDDO
ENDIF
```

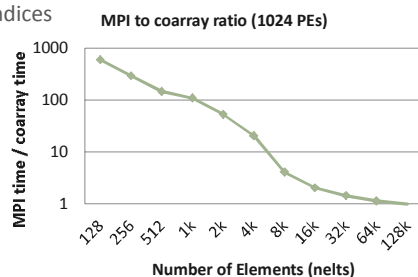
## Remote gather: coarray implementation (get)

- Image 1 gets the values from the other images

```
IF (myimg.eq.1) THEN
 DO i=1,nelts
 pe=(index(i)-1)/nloc+1
 offset=MOD(index(i)-1,nloc)+1
 caf_buffer(i)=caf_table(offset)[pe]
 ENDDO
ENDIF
```

## Remote gather: coarray vs MPI

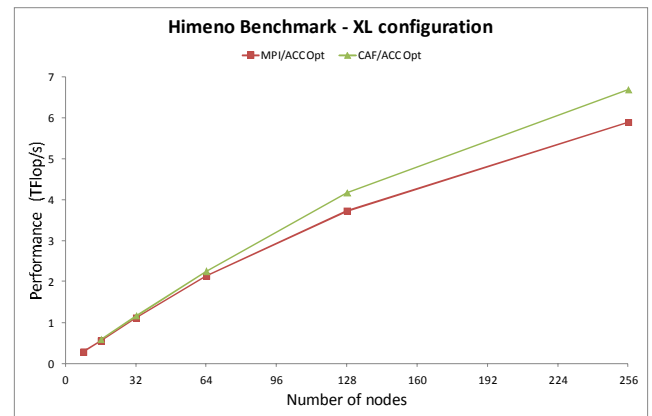
- Coarray implementations are much simpler
- Coarray syntax allows the expression of remote data in a natural way – no need of complex protocols
- Coarray implementation is orders of magnitude faster for small numbers of indices



## HIMENO

- HIMENO Halo-Swap benchmark
- Uses Jacobi method to solve Poisson’s equation
- Looked at a distributed implementation for GPUs
- When distributed this gives a stencil computation and halo-swap communication.
  - Used draft OpenMP GPU directives stencil computation
  - used MPI or coarrays for halo-swap between processes
- Coarray code for halo-swap was simple and was best performing of the optimized versions
- There is still scope to optimize the coarray version (reduce extra data copy)

## HIMENO



## Gyrokinetic Fusion Code

- Particle in Cell (PIC) approach to simulate motion of confined particles
- Motion caused by electromagnetic force on particle
- Many particles stay in a cell for small timestep but some don’t
- Timestep chosen to limit travel to 4 cells away
- Departing particles stored in a buffer and when this is full the data is sent to the neighboring cell’s incoming buffer
- Force fields recomputed once particles are redistributed

- Coarrays used to avoid coordinating the receive of the data
- SC11 paper

## References

- “Cray’s Approach to Heterogeneous Computing”, R. Ansaloni, A. Hart, Parco 2011 (to appear).
  - “The Himeno benchmark”, Ryutaro Himeno, [http://accr.riken.jp/HPC\\_e/himenobmt\\_e.html](http://accr.riken.jp/HPC_e/himenobmt_e.html).
- “Multithreaded Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms”, Robert Preissl, Nathan Wichmann, Bill Long, John Shalf, Stephane Ethier, Alice Koniges, SC11 best paper finalist

## References

- <http://lacs.rice.edu/software/caf/downloads/documentation/nrRAL98060.pdf> - Co-array Fortran for parallel programming, Numrich and Reid, 1998
- <ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf> "Coarrays in the next Fortran Standard", John Reid, April 2010
- Ashby, J.V. and Reid, J.K (2008). Migrating a scientific application from MPI to coarrays. CUG 2008 Proceedings. RAL-TR-2008-015  
See <http://www.numerical.rl.ac.uk/reports/reports.shtml>
- <http://upc.gwu.edu/> - Unified Parallel C at George Washington University
- <http://upc.lbl.gov/> - Berkeley Unified Parallel C Project



49

## Tutorial Wrapup

Remember our first Motivation slide?

- Fortran now supports parallelism as a full first-class feature of the language
- Changes are minimal
- Performance is maintained
- Flexibility in expressing communication patterns

We hope you learned something and have success with coarrays in the future

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 8 | 4 | 3 | 4 | 2 | 5 | 6 | 7 |
| 9 | 1 | 7 | 6 | 1 | 4 | 2 | 3 | 5 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 5 | 6 | 8 |
| 8 | 6 | 1 | 5 | 3 | 7 | 2 | 9 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 5 | 3 |
| 3 | 4 | 5 | 2 | 6 | 1 | 7 | 9 | 8 |



50

## Acknowledgements

The material for this tutorial is based on original content developed by EPCC of The University of Edinburgh for use in teaching their MSc in High-Performance Computing. The following people contributed to its development:

Alan Simpson, Michele Weiland, Jim Enright and Paul Graham

The material was subsequently developed by EPCC and Cray to form this tutorial with contributions from the following people:

David Henty, Alan Simpson (EPCC)  
Harvey Richardson, Bill Long, Roberto Ansaloni,  
Jef Dawson, Nathan Wichmann (Cray)

This material is Copyright © 2011  
by The University of Edinburgh and Cray Inc.





# EXERCISE ASSIGNMENTS

# Practicalities

## Computing servers

We will use CSC's Cray supercomputer Louhi for the exercises. Log onto Louhi using the provided `trngXX` username and password, e.g.

```
% ssh -X trng10@louhi.csc.fi
```

Alternatively, feel free to use the local workstations or your own Linux/Mac laptop and GNU compiler, but for the Co-Array Fortran features we will need to use the Cray Fortran compiler. To enable the Cray compiler, we need to do (once on Louhi)

```
% module swap PrgEnv-pgi PrgEnv-cray
```

Other compilers (GNU, PGI, Pathscale, and Intel) are also available in CSC's computing servers.

The compiler can be changed via (for example)

```
% module swap PrgEnv-cray PrgEnv-gnu
```

For editing Fortran program files you can use e.g. Emacs editor with or without (the option `-nw`) X-Windows:

```
emacs -nw prog.f90
```

```
emacs prog.f90
```

Also other popular editors (vim, nano) are available.

## Simple compilation

Example program `test.f90`

```
program test
 implicit none
 integer, parameter :: result = 42
 print *, 'hello world!'
 print *, 'the result is', result
end program test
```

Compilation and execution are done via the **ftn** wrapper and the **aprun** scheduler:

```
% ftn test.f90 -o test
```

```
% aprun -n 1 ./test
```

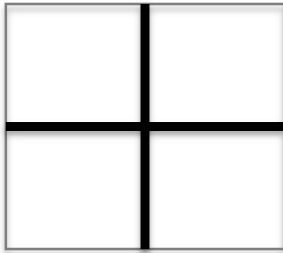
```
Hello world!
```

```
The result is 42
```

# Fortran 95/2003 exercises

## 1. Playing around with control structures and arrays

- a. Declare an integer array of dimensions 100 by 100 and initialize it to zero otherwise but where the first index is equal to 50 or the second index is equal to 50 the array elements get a value of 1.

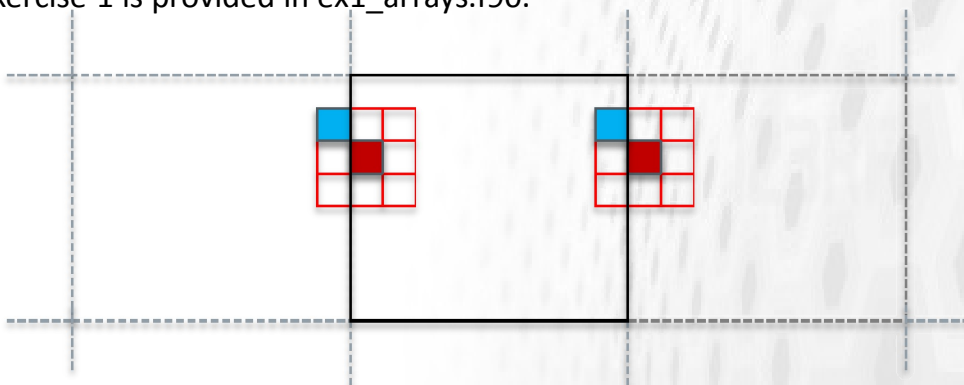


Value 1 on the black rows, value 0 elsewhere

- b. Then initialize another array with the same dimensions, but its values are computed from the first array such that
- The cells with exactly two elements equal to 1 in the first array get the same value in the new array as in the first array.
  - Any cell with exactly three neighbors with value 1 gets the value 1 in the new array.
  - Otherwise, i.e. new array cells with less than two or more than three neighbors with value 1 in the old array get a value 0.

At this stage, just run the indices from 2 to 99, i.e. not referencing to the boundaries at all.

- c. Modify the program from exercise 1b such that the array becomes periodic – that is, the boundaries depend on the cells on the other side of the array. The solutions for all three items of Exercise 1 are provided in ex1\_arrays.f90.



## 2. Getting acquainted with procedures

- a. Modify the program such that you can produce new arrays iteratively, i.e. taking the array from the previous iteration and obtaining a new array by applying the same rules for it.
- b. Modify the exercise 2a so that it uses a function or a subroutine to produce the new array.
- c. Change the initialization of the board such that the board starts from a random configuration. The intrinsic procedure is called `RANDOM_NUMBER`. Wrap also this board initialization into its own procedure. The solutions for all three items of Exercise 2 are provided in ex2\_procedures.f90.

# Fortran 95/2003 exercises

## 3. Game of Life

The *Game of Life* (GoL) is a cellular automaton devised by John Horton Conway in 1970, see [http://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](http://en.wikipedia.org/wiki/Conway's_Game_of_Life).

You can compile a reference executable (since the file `ex3_gol.f90` will contain the solution) with

```
% ftn -o gol gol_io.f90 ex3_gol.f90
```

Run the program of e.g. 200x200 board for 100 iterations. With the command `xview` or `eog` you can view the images (.pbm) and see how the automaton looks like after those. You can also animate the board development by first using `convert` as

```
% convert -delay 40 -geometry 512x512 life_*.pbm life.gif
```

(on a single line) and then displaying the animation with

```
% animate life.gif
```

- See the file `ex3_gol.f90`, get acquainted with the program and complete the missing parts of the code (search: "TODO"). Refer back to assignments 1 and 2.
- Experiment, how the game evolves if you replace the starting pattern ('plus', c.f. exercise 1a) to a random one (c.f. Exercise 2c).
- Modify the GoL program such that the board is manipulated through a derived datatype `GoL_board`, which contains the actual board, its dimensions as well as how many iterations it has gone through. No solution has been prepared.
- Now we will examine the I/O module of the Game of Life program. It visualizes the board in the netpbm image format, see [http://en.wikipedia.org/wiki/Netpbm\\_format](http://en.wikipedia.org/wiki/Netpbm_format). Implement the writing of the board as pbm images - or in some other image format if you want to go your own way. Consult the `gol_io.f90` when in trouble. *Shortcut*: study the **draw** subroutine in `gol_io.f90` and make sure you understand the piece of code.
- Modify the program such that the user input is read directly from the command line instead of parsing, i.e. the program is launched as `./gol (# iterations) (board height) (board width)`, for example

```
% ./gol 200 100 100
```

The answer is provided in `gol_io.f90`.

# More bonus exercises

## Fortran Quiz

- a. Are the following Fortran statements written correctly?
- ```
character_string = 'Awake in the morning,  
    & asleep in the evening.'  
x = 0.4-6  
answer = 'True & false'  
low-limit = 0.0005E10  
y = E6
```
- b. Are the following declarations legitimate in Fortran?
- ```
DOUBLE :: x
CHARACTER(LEN=*), PARAMETER :: "Name"
REAL :: pi = 22/7
REAL :: x = 2., y = -3
REAL :: pii = 22.0/7.0
REAL x = 1.0
```
- c. What are the iteration counts of the following DO loops, the values of loop variable i inside the loop, and the value of the loop variable after the DO construct?
- ```
DO i = 1, 5  
DO i = 5, 0, -1  
DO i = 10, 1, -2  
DO i = 0, 30, 7  
DO i = 3, 2, 1
```

Derived types

- a. Declare the derived type which can save the birth date in the form:
21 01 1990
This derived type thus contains three integers, which have different KIND values:
SELECTED_INT_KIND(2) and SELECTED_INT_KIND(4).
- b. Add the field the for a name to the derived type. Write a function, which returns the name and date in a character string in the following form
Charlie Brown (01.01.1999)

Recursion

Write a recursive function, which calculates "Tribonacci numbers":

$$\begin{cases} x_1 = 1, & x_2 = 1, & x_3 = 1, \\ x_n = x_{n-1} + x_{n-2} + x_{n-3}. \end{cases}$$

Calculate x_{12} . Carry out the computation also using a loop structure.

Parallel Programming with Fortran Coarrays: Overview of Exercises

Delivered at PRACE Advanced Training Centre,
CSC – IT Center for Science Ltd, Finland,
September 13, 2012

David Henty, Alan Simpson (EPCC)
Harvey Richardson, Bill Long (Cray)



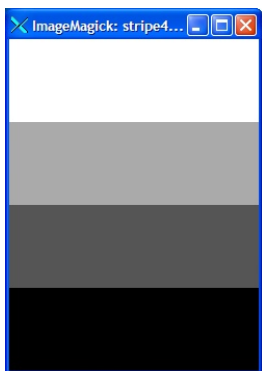
Exercise 1

- Hello world example
 - check you can log on, compile, submit and run
- Writing arrays as pictures
 - declare and manipulate coarrays
 - write out arrays in PGM picture format
 - view them using **display** from ImageMagick
 - use both remote reads and remote writes



2

Sample output on 4 images



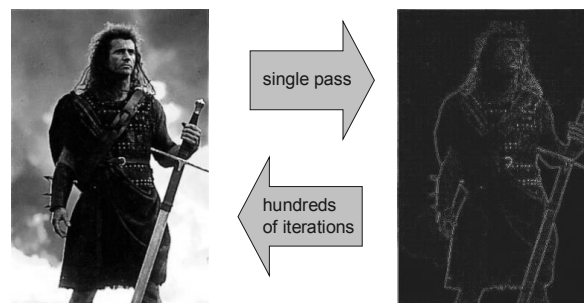
3

Exercise 2

- Perform simple edge detection of features in a picture
 - halo communication between 1D grid of images
- Reconstruct picture from supplied edges
 - an iterative algorithm
 - computationally intensive so worth parallelising
- Terminate based on some stopping criterion
 - requires global sums
- Use global or point-to-point synchronisation
- Look at scalability



Edge detection and picture reconstruction



5

Exercise 3 (unlikely to get here today!)

- Decompose picture across a 2D grid of images
 - using multiple codimensions



Documentation

- Full instructions in exercise notes
 - PDF copy in doc/ subdirectory
- **Go at your own pace**
 - no direct dependencies between practicals & lectures
 - each exercise follows on from the last
- If you're not sure what to do or if you have any other questions then please ask us!

