Fortran Modernisation Workshop

Numerical Algorithms Group

Wadud Miah

Jon Gibson

Experts in numerical software and
High Performance Computing

Copyright Statement (1)

- © The Numerical Algorithms Group Limited, 2018
- All rights reserved. Duplication of this presentation in printed form or by electronic means for the private and sole use of the delegate is permitted provided that the individual copying the document is not:
  - selling or reselling the documentation;
  - distributing the documentation to others;
  - using it for the purpose of critical review, publication in printed form or any electronic publication including the Internet without the prior written permission of the copyright owner.
- The copyright owner gives no warranties and makes no representations about the contents of this presentation and specifically disclaims any implied warranties or merchantability or fitness for any purpose;

Copyright Statement (2)

- The copyright owner reserves the right to revise this presentation and to make changes from time to time in its contents without notifying any person of such revisions or changes.

The Numerical Algorithms Group

- Experts in Numerical Computation and High Performance Computing
- Founded in 1970 as a co-operative project out of academia in UK
- Operates as a commercial, not-for-profit organization
  - Funded entirely by customer income
- Worldwide operations
  - Oxford & Manchester, UK
  - Chicago, US
  - Tokyo, Japan
- Over 3,000 customer sites worldwide
- NAG's code is embedded in many vendor libraries

Experts in Numerical Computation and HPC

- NAG Library
- NAG Fortran Compiler
- Algorithmic Differentiation
- Bespoke numerical solvers and custom adjoints
- Grid/cloud execution framework
- Code modernization and parallelization
- Technology evaluation and benchmarking
- HPC advice and procurement assistance

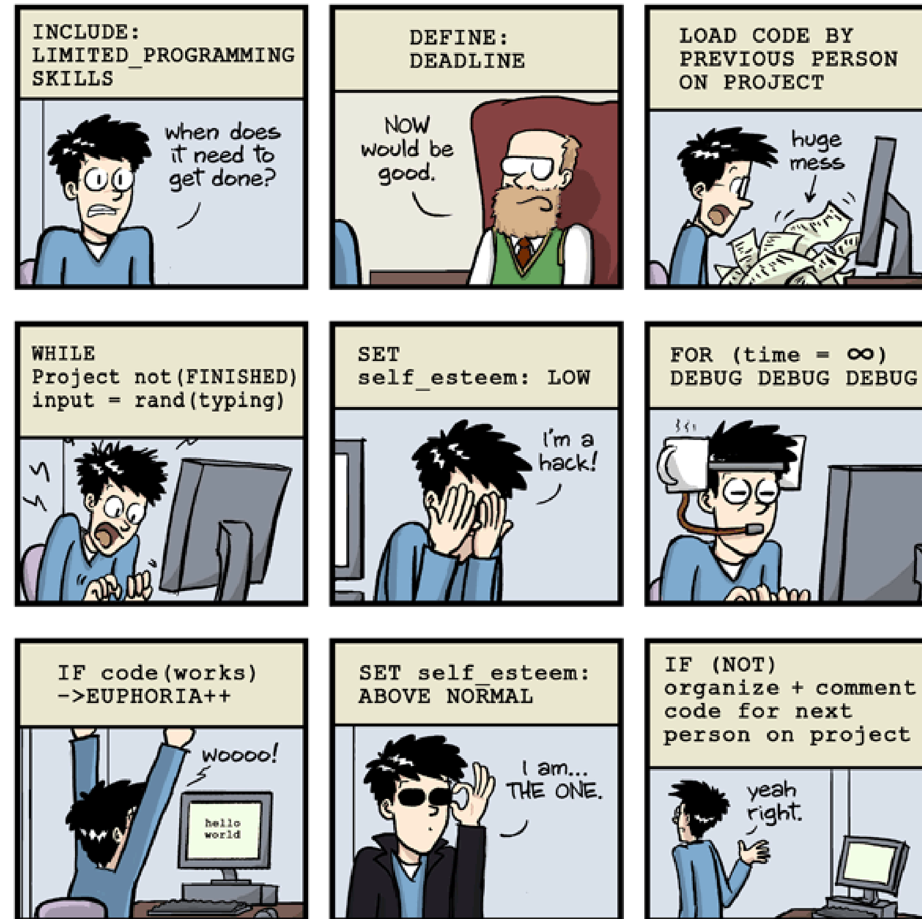
NAG and Fortran (1)

- 50 years of good practice in scientific computing and HPC to build robust, portable and performing numerical code;
- Fortran is the programming language of choice to develop the kernel of the NAG Numerical Library;
- NAG develop their own Fortran compiler, led by Malcolm Cohen;
- Malcolm Cohen is also member of the Fortran standards committee and developed the first Fortran 90 compiler;

NAG and Fortran (2)

- He is the co-author of the famous "Modern Fortran Explained" book;
- NAG Compiler test suite is being used by a number of other compiler writers to validate their own software;
- NAG are also contributing to the language through this workshop.

Programming by Scientists



http://phdcomics.com

Day One Agenda

- History of Fortran;
- Source code formatting and naming conventions;
- Source code documentation using comments;
- Memory management and pointers;
- Fortran strings and Fortran modules and submodules;
- Numerical, user defined data types and designing good APIs;
- Refactoring legacy Fortran;
- Makefile. Serial NetCDF, serial HDF5, and PLplot;
- Day one practical;
- Supplementary material at www.nag.co.uk/content/fortran-modernization-workshop

History of Fortran (1)

- Fortran or Fortran I contained 32 statements and developed by IBM – 1950;
- Fortran II added procedural features – 1958;
- Fortran III allowed inlining of assembly code but was not portable – 1958;
- Fortran IV become more portable and introduced logical data types – 1965;
- Fortran 66 was the first ANSI standardised version of the language which made it portable. It introduced common data types, e.g. integer and double precision, block IF and DO statements – 1966;

History of Fortran (2)

- Fortran 77 was also another major revision. It introduced file I/O and character data types – 1977;
- Fortran 90 was a major step towards modernising the language. It allowed free form code, array slicing, modules, interfaces and dynamic memory amongst other features – 1990;
- Fortran 95 was a minor revision which includes pointers, pure and elemental features – 1995;
- Fortran 2003 introduced object oriented programming. Interoperability with C, IEEE arithmetic handling – 2003;

History of Fortran (3)

- Fortran 2008 introduced parallelism using CoArrays and submodules – 2008;
- Fortran 2018 improved the CoArray features by adding collective subroutines, teams of images, listing failed images and atomic intrinsic subroutines – 2018;
- Most compilers, to date, support Fortran 77 to Fortran 2008. See [1] and [2] for further details;
- This workshop will be discussing Fortran 90, 95, 2003, 2008 and 2018 also known as *modern Fortran.*

[1] http://www.fortran.uk/fortran-compiler-comparisons-2015/
[2] http://www.fortranplus.co.uk/resources/fortran_2003_2008_compiler_support.pdf

Fortran Standards Committee

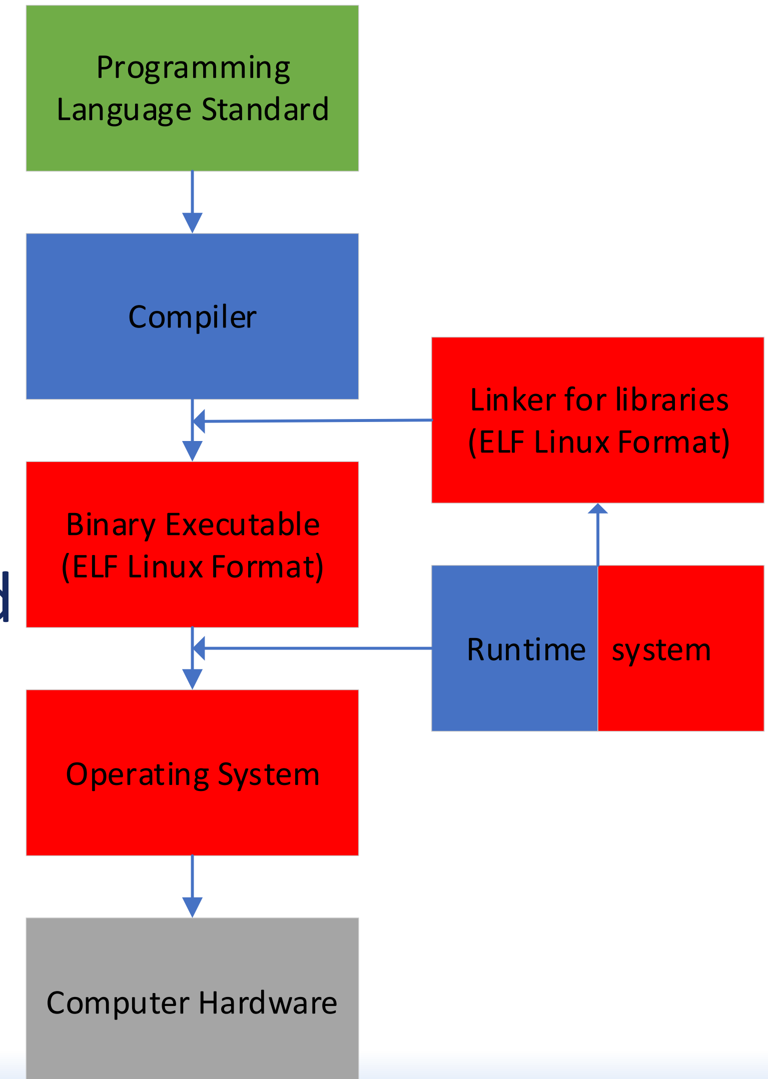
- The Fortran Standards Committee members are comprised of industry, academia and research laboratories;
- Industry: IBM, Intel, Cray, Numerical Algorithms Group (NAG), Portland Group (Nvidia), British Computer Society, Fujitsu;
- Academia: New York University, University of Oregon, George Mason University, Cambridge University and Bristol University;
- Research laboratories: NASA, Sandia National Lab, National Center for Atmospheric Research, National Propulsion Laboratory, Rutherford Appleton Laboratory (STFC)

Fortran Compilers

- Fortran compiler vendors include Intel, PGI (Nvidia), NAG, Cray, GNU, Flang, IBM, Lahey, NEC and Arm;
- Fortran compiler vendors then implement the agreed standard;
- Some vendors are quicker than others in implementing the Fortran standard;
- Large choice of compilers, each with their strengths and weaknesses. No "best" compiler for all situations, e.g. portability to performance;
- Some have full or partial support of the standard.

Compiler Characteristics (1)

- **The compiler is not the language**. It is an application like any other and has bugs. It is more thoroughly tested than other applications;
- How well it implements the language standard varies across compilers;
- It uses the Linux linker to create executables and requires a runtime system to execute by the operating system;
- Runtime systems are supplied by the compiler and the operating system;



Compiler Characteristics (2)

- **Performance** - how well is the code optimised on the target architecture;
- **Correctness** - does it detect violations of the language specification ideally at compilation or at runtime? Does it print helpful error messages when a violation is detected?
- **Features** - does it support newer standards, e.g. 2008?
- **Compilation speed** - related to all the above;
- Additional software development tools bundled with the compiler;
- All the above characteristics should be considered when using a compiler and not just one, e.g. performance;

Compiler Characteristics (3)

- Compiler optimisations are compiler dependent. The Fortran standard does not specify how the language should be implemented, e.g. whether array operations are vectorised;
- The standard is written so that it allows compilers to optimise the code, but performance across compilers can vary considerably;
- There is no guarantee a newer compiler will run your code faster. It could run slower;
- Only guarantee that compilers try to give is that it produces the correct answer given a **valid** Fortran code.

Evolution of the Fortran Standard

- As the standard evolves, language features get **obsoleted** and then **deleted** from the standard;
- When a feature is obsoleted, it is **marked** for deletion and replacement feature is incorporated;
- In one of the next revisions, the feature is permanently **deleted**;
- Some compilers will continue to support deleted features or might completely remove them;
- To ensure your code is fully portable, it is important to keep it up to date with the language standard, i.e. modernise your code!

Compiler Extensions and the Standard (1)

- Some compilers provide extensions to the language which are not part of the official language standard, e.g. CUDA Fortran;
- Some are useful in that they provide extra features or improve performance;
- However, they are usually unique to that compiler (or a few compilers) and there is no guarantee that the compiler vendor will continue to support it;
- Or worse, the compiler vendor might no longer exist which will cause serious problems when attempting to use another compiler;

Compiler Extensions and the Standard (2)

- This will pose serious portability issues, so be careful when using compiler extensions or just using one compiler;
- To increase portability of your code, strictly adhere to the language standard;
- Compiler vendors try to adhere the language standard and they are reasonably successful in doing so.

Code Structure

- Modularise your code so that components can be re-used and better managed by a team of developers;
- *Write code so that it can be tested*;
- Use `implicit none` so that *all variables have to be explicitly defined*;
- Use whitespace to make your code readable for others and for yourself;
- Use *consistent* formatting making it easier to read the entire code;
- *Agree on a formatting standard for your team so that you can read each other's code in a consistent manner.*

Coding Style Suggestions (1)

- Use lower case for all your code[1], including keywords and intrinsic functions. IDEs now highlight such identifiers;
- Capitalise first character of subroutines and functions, and use spaces around arguments:

```
a = VectorNorm( b, c ) ! Or use underscore
a = Vector_norm( b, c )
```

- Use lower case for arrays *and no spaces*:

```
a = matrix(i, j)
```

- The difference between function and array references are clearer;

[1]Exceptions apply

Coding Style Suggestions (2)

- Use *two character spaces when indenting blocks of code* and increase indentation with nested blocks, and name your block statements:

```
CELLS: do i = 1, MAX_CELLS
  EDGE: if ( i == MAX_CELLS ) then
    vector(i) = 0.0
  else
    vector(i) = 1.0
  end if EDGE
end do CELLS
```

- Name large blocks containing sub-blocks as shown above;

Coding Style Suggestions (3)

- Use spaces around IF statement parentheses:

```
SCALE: if ( i <= MAX_CELLS ) then
  vector(i) = alpha * vector(i)
end if SCALE
```

- Use symbolic relational operators:

Old Fortran	New Fortran	Description
.GT.	>	greater than
.GE.	>=	greater than or equal to
.LT.	<	less than
.LE.	<=	less than or equal to
.NE.	/=	not equal to
.EQ.	==	equal to

Coding Style Suggestions (4)

- Always use the double colon to define variables:

```
real :: alpha, theta
integer :: i, j, k
```

- Use square brackets to define arrays and use a digit on each side of the decimal point:

```
vec = (/ 0.0, 1.0, 2.0, 3.0 /)    ! old Fortran
vec = [ 0.0, 1.0, 2.0, 3.0 ]      ! Fortran 2003
```

- Separate keywords with a space:

enddo	end do
endif	end if
endfunction	end function
endmodule	end module
selecttype	select type

nag

Coding Style Suggestions (5)

- Use a character space around mathematical operators and use brackets to show precedence - this can also aid compiler optimization:

```
alpha = vector(i) + ( beta * gamma )
```

- Always use character spaces after commas:

```
do j = 1, Nj
   do i = 1, Ni
      matA(i, j) = matA(i, j) + matB(i, j)
   end do
end do
```

- Remember that Fortran is column-major, i.e. `a(i, j), a(i+1, j), a(i+2, j)` are contiguous;

Coding Style Suggestions (6)

- Since Fortran is column-major, ensure the contiguous dimension is passed to procedures:

```
call array_calculation( A(:, :, k), alpha )
```

- This will work, but will be slow:

```
call array_calculation( A(i, :, :), alpha )
```

- Capitalise names of constants:

```
integer, parameter :: MAX_CELLS = 1000
```

nag

Using Comments

- Use comments to describe code that is not obvious;
- Indent comments with block indenting;
- Use comments on the line before the code:

```
! solve the shock tube problem with UL and UR
call Riemann( UL, UR, max_iter, rtol, dtol )
```

- Always comment the beginning of the file with:
  a) purpose of code. Include LaTeX code of equation;
  b) author and email address;
  c) date;
  d) application name;
  e) any licensing details.

Naming Conventions (1)

- Use function, subroutine and variables names that are meaningful to your scientific discipline;
- The wider the scope a variable has, the more meaningful it should be;
- When using Greek mathematical symbols, use the full name, e.g. use `alpha` instead of `a`. Good names are self-describing;
- For functions and subroutines, use verbs that describe the operation:

```
Get_iterations( iter )
Set_tolerance( tol )
Solve_system( A, b, x )
```

nag

Naming Conventions (2)

- Avoid generic names like `tmp` or `val` even in functions/subroutines that have a scope outside more than one block;
- Loops variables such as `i`, `j`, `k`, `l`, `m`, `n` are fine to use as they are routinely used to describe mathematical algorithms;
- Reflect the variables as much as possible to the equations being solved; so for $p = \rho R T$:

```
p = rho * R * T
```

Naming Conventions (3)

- In functions and subroutines use the `intent` keyword when defining dummy arguments;
- If using subroutines from third-party libraries, capitalise the name, e.g. `MPI_INIT( ierr )`

Short Circuiting IF Statements

- Fortran does not short circuit IF statements:

```
if ( size( vec ) == 10 .and. vec(10) > eps ) then
  ! [ ... ]
end if
```

- The above could result in a segmentation fault caused by array out of bounds access. Instead, use:

```
if ( size( vec ) == 10 ) then
  if ( vec(10) > eps ) then
    ! [ ... ]
  end if
end if
```

Fortran 90 Arrays (1)

- Fortran 90 arrays can be defined using:

```
real, dimension(1:10) :: x, y, z
```

- Scalar operations can be applied to multi-dimensional data:

```
x(1:10) = y(1:10) + z(1:10)
```

- This can be parallelised using OpenMP:

```
!$omp parallel workshare shared(x,y,z)
   x(:) = y(:) + z(:)
!$omp end parallel workshare
```

- Use `lbound( )` and `ubound( )` intrinsic functions to get lower and upper bound of multi-dimensional arrays;

Fortran 90 Arrays (2)

- When referring to arrays, use the brackets to indicate the referencing of an array, e.g.

```
result(:) = vec1(:) + vec2(:)
call Transpose( matrix(:, :) )
```

- Array operations are usually vectorised by your compiler. Check Intel Fortran compiler vectorisation report using the flags:

```
-qopt-report-phase=vec,loop -qopt-report-file=stdout
```

- You can also create HTML reports for continuous integration systems:

```
-qopt-report-annotate=html
```

Fortran 90 Arrays (3)

- Using do loops for array assignments can create bugs;
- Spot the bug below:

```
real, dimension(3) :: eng, aero
do i = 1, 3 ! 1 = port, 2 = centre, 3 = starboard
  aero = eng(i)
end do
! simplified version. always use brackets to show array
! operations
aero(:) = eng(:)
```

- Array operations are more likely to vectorise than their loop equivalents;

nag

Fortran 90 Arrays (4)

- You can also use the array notation in the GNU debugger:

```
$ gdb vec_test.exe
(gdb) break 1
Breakpoint 1, vec_test () at vec_test.f90:7
7    a(:) = 1.0
(gdb) print a(1:3)
$1 = (1, 1, 1)
(gdb) print a(:)
$2 = (1, 1, 1, 1, 1, 1, 1, 1)
(gdb)
```

Fortran 90 Array Masking (1)

- Array operations can also be applied to elements that satisfy a condition:

```
where ( uu(:) > 0 ) u(:) = v(:) / uu(:)
where ( val(:) > 0 )
  res(:) = log( val(:) )
elsewhere
  res(:) = abs( val(:) )
end where
```

nag

Fortran 90 Array Masking (2)

- The following intrinsic functions also take a mask argument:

  `all( ), any( ), count( ) maxval( ), minval( ), sum( ), product( ), maxloc( ) and minloc( )`

- For example:

```
sval = sum( val(:), mask = val(:) > 1.0 )
```

- Masked array operations can still be vectorised by using the Intel Fortran compiler flag `-vec-threshold`*n* where *n* is between 0 and 100;

nag®

Fortran 90 Array Masking (3)

- If 0, loop gets vectorised always and if 100, compiler heuristics will determine level of vectorisation;
- Use the `-align array64byte` flag to align double precision arrays on vector boundaries;
- Array operations are one of the strengths of the Fortran language which modern scripting languages have.

nag

More Array Operations

- The intrinsic function `pack` collates array elements:

```
vec(:) = [ 1, 0, 0, 5, 0 ]
pack( vec(:), vec(:) /= 0 ) != [ 1, 5 ]
```

- The intrinsic function `transpose` flips a two-dimensional array:

```
mat(:, :) = reshape( [ 1, 2, 3, 4 ], shape( mat ) )
print *, mat, transpose( mat ) ! prints 1, 2, 3, 4 and
1, 3, 2, 4
```

Famous Gauss-Seidel Method

```
do iter = 1, num_iterations
  do j = 2, Nj - 1
    do i = 2, Ni - 1
       A_new(i, j) = outside(i, j) * A(i, j) + inside(i, j) * &
                     0.25_DP * (A(i + 1, j) + A(i - 1, j) + &
                                A(i, j + 1) + A(i, j - 1))
    end do
  end do

  A(:, :) = A_new(:, :)
end do
```

Array Version of Gauss-Seidel Method

```
do iter = 1, num_iterations
   A_new(:, :) = outside(:, :) * A(:, :) + inside(:, :) * 0.25_DP * &
                           ( cshift(A(:, :), dim = 1, shift = 1 )  + &
                             cshift(A(:, :), dim = 1, shift = -1 ) + &
                             cshift(A(:, :), dim = 2, shift = 1 )  + &
                             cshift(A(:, :), dim = 2, shift = -1 ))

  A(:, :) = A_new(:, :)
  if ( all( abs( A_new(:, :) - A(:, :)) < epsilon )) exit
 end do
```

nag®

Derived Data Type Names (1)

- When defining derived types, use the `t` suffix:

```
type point_t
  real :: x, y, z
end type point_t
type(point_t) :: p1, p2, p3
```

- For assignment, you can use two methods:

```
p1 = point_t( 1.0, 1.0, 2.0 ) ! or
p1%x = 1.0
p1%y = 1.0
p1%z = 2.0
```

nag

Derived Data Type Names (2)

- For pointers, use the `p` suffix:

```
type(point_t), pointer :: centre_p
centre_p => p1
```

- Can have a type within a type:

```
type square_t
  type(point_t) :: p1
  type(point_t) :: p2
end type square_t
type(square_t) :: s1, s2
s1%p1%x = 1.0
```

Extensible Data Types

```
type, extends(point_t) :: point4_t
  real :: t
end type point4_t
type(point4_t) :: p1
! x = 1.0, y = 2.0, z = 3.0, t = 4.0
p1 = point4_t( 1.0, 2.0, 3.0, 4.0 )
```

Parameterised Data Types

```
type matrix( k, Ni, Nj )
  integer, kind :: k = REAL32 ! default precision
  integer, len :: Ni, Nj
  real(kind=k), dimension(Ni,Nj) :: matrix
end type matrix
! double precision
type(matrix(k=REAL64,Ni=10,Nj=10)) :: A
type(matrix(Ni=10,Nj=10)) :: B  ! single precision
A%matrix(:, :) = 1.0_REAL64
```

nag

Derived Data Type I/O

- Derived data types can also written to a file in a single statement:

```
p1 = point_t( 1.0, 2.0, 3.0 )
p2 = point_t( 2.0, 3.0, 4.0 )
print *, 'Free format output ', p1
print '(A,3F10.2)', 'Formatted output', p2
```

- Output is:

```
 Free format output    1.0000000   2.0000000   3.0000000
Formatted       2.00      3.00      4.00
```

nag

Array of Derived Data Types

```
type point_t
  real :: x, y, z
end type point_t
type(point_t), dimension(1:100) :: points

do i = 1, 100
  points(i)%x = 1.0; points(i)%y = 1.0
  points(i)%z = 1.0
end do
```

- *The above code will not be vectorised.*

Derived Data Types With Arrays

```
type point_t
  real, dimension(1:100) :: x, y, z
end type point_t
type(point_t) :: points
points%x(:) = 1.0
points%y(:) = 1.0
points%z(:) = 1.0
```

- *The above code will be vectorised.*

Function and Subroutine Arguments (1)

- Always use the `intent` keyword to precisely define the usage of the dummy arguments in functions and subroutines;
- When an argument needs to be read only by a procedure:

```
subroutine Solve( tol )
  real, intent(in) :: tol
end subroutine Solve
```

- When an argument needs to be written only by a procedure:

```
real, intent(out) :: tol
```

Function and Subroutine Arguments (2)

- For an argument that needs to be read and written by a subroutine or function:

```
real, intent(inout) :: tol
```

- Note that Fortran arguments are by reference. They are not copied so subroutine or function invocations are quicker and use less stack memory;
- If arguments are misused, this will be flagged during compilation which will help you write correct code;
- Recommendation is to list the `intent` attribute last.

Function and Subroutine Arguments (3)

- Recommendation is to *list all dummy arguments first followed by local variables*;
- For pointer arguments, scoping is only relevant to association:

```
subroutine sub1( x_p, x_t )
  real, pointer, intent(in) :: x_p
  real, intent(in) :: x_t
  x_p = x_t    ! valid
  x_p => x_t   ! invalid
end subroutine sub1
```

Intent of Derived Data Type (1)

```
type(point_t) :: p1
p1%x = 1.0; p1%y = 2.0
call init( p1 )
! p1%y could be undefined here
subroutine init( p1 )
  type(point_t), intent(out) :: p1
  p1%x = 2.0 ! p1%y is not being assigned
end subroutine init
```

Intent of Derived Data Type (2)

- Better to use `intent(inout)` instead of `intent(out)` as shown below:

```
subroutine init( p1 )
  type(point_t), intent(inout) :: p1

  p1%x = 2.0
end subroutine init
```

Command Line Arguments

- Fortran 2003 allows the retrieval of command line arguments passed to the code:

```
character(len=60) :: arg
integer :: i, len, ierr
do i = 1, command_argument_count( )
  call get_command_argument( i, value = arg, length = &
                     len, status = ierr )
  write (*,*) i, len, ierr, trim( arg )
end do
```

Avoiding Go To Statements

- Go to statements are sometimes useful but they are discouraged because they are generally difficult to manage;
- Instead use `cycle` or `exit` statements in loops:

```
OUTER: do i = 1, Ni
  INNER: do j = 1, Nj
    ! cycle will move onto the next j iteration
    if ( condition1 ) cycle INNER
  end do INNER
  ! exit will break out of the OUTER loop
  if ( condition2 ) exit OUTER
end do OUTER
```

Fortran Block Statements

- Fortran block statements can also be used to avoid go to statements;

```
subroutine calc( )
MAIN1: block
  if ( error_condition ) exit MAIN1

  return ! return if everything is fine
end block MAIN1
! add exception handling code here
end subroutine calc
```

Memory Management (1)

- Fortran 90 introduced dynamic memory management which allows memory to be allocated at run time;
- Use dynamic memory allocation if your problem size will vary and specify the start index:

```
real, dimension(:), allocatable :: vector
character(len=120) :: msg
allocate( vector(1:N), stat = ierr, errmsg = msg )
```

- Always give the first index. The `errmsg` argument is Fortran 2008;
- The integer `ierr` is zero if allocation is successful. If this is non-zero, then check the error message variable `msg`;

Memory Management (2)

- Then deallocate when not required:

```
deallocate( vector, stat = ierr )
```

- *Remember to deallocate if using pointers* – if not, it could cause memory leaks[1];
- *Instead of using pointers, use the* `allocate` *keyword which makes variables easier to manage for both the developer and the compiler. The Fortran language will automatically deallocate when variable is out of scope;*

[1]Use NAG compiler, Valgrind or RougeWave MemoryScape to debug memory problems

Memory Management (3)

- Can use the `allocated( array )` intrinsic function to check whether memory has been allocated;
- *You cannot allocate twice (without deallocating) which means you will not suffer from memory leaks!*

nag

Memory Optimizations

- *Try to use unit stride when referencing memory*, e.g. do not use:

```
mesh(1:N:4)
```

- Instead refer to *contiguous* memory:

```
mesh(1:N)
```

- The above unit stride array allows the compiler to *vectorise* operations on arrays;
- In addition, *it allows better cache usage*, therefore optimising your memory access and computation;
- Passing unit stride arrays to subroutines and functions are quicker and use less memory.

Array Arguments - Explicit Shape Arrays

```
subroutine init( vec, arg )
  real, intent(out), dimension(100) :: vec ! contiguous
  real, intent(in) :: arg
  vec(:) = arg
end subroutine init
```

Array Arguments - Adjustable Arrays

```
subroutine init( vec, n, arg )
  integer, intent(in) :: n
  real, intent(out), dimension(n) :: vec ! contiguous
  real, intent(in) :: arg
  vec(:) = arg
end subroutine init
```

Array Arguments - Assumed Size Arrays

```
subroutine init( vec, n, arg )
  integer, intent(in) :: n
  real, intent(out), dimension(*) :: vec  ! contiguous
  real, intent(in) :: arg
  vec(1:n) = arg
  ! vec(:) is illegal. dimension is lost
end subroutine init
```

Array Arguments - Allocatable Arrays

```
subroutine init( vec, n, arg )
  integer, intent(in) :: n
  real, dimension(:), allocatable :: vec
                                 ! not contiguous
  real, intent(in) :: arg

  allocate( vec(1:n) ) ! contiguous
  vec(1:n) = arg
end subroutine init
```

Array Arguments - Assumed Shaped Arrays

```
subroutine init( vec, arg )
  real, dimension(:), intent(out) :: vec
                        ! not contiguous
  real, intent(in) :: arg

  vec(:) = arg
end subroutine init
```

nag

Array Arguments - Pointer Argument (1)

```
subroutine init( vec, arg )
  real, dimension(:), pointer, intent(in) :: vec
                             ! not contiguous
  real, intent(in) :: arg

  if ( associated( vec )) vec(:) = arg
end subroutine init
```

Array Arguments - Pointer Argument (2)

```
real, dimension(1:100), target :: vec
real, dimension(:), contiguous, pointer :: vec_p

vec_p => vec; call init( vec_p, 1.0 )
subroutine init( vec, arg )
  real, dimension(:), pointer, contiguous, intent(in) :: vec
  real, intent(in) :: arg

  if ( associated( vec )) vec(:) = arg
end subroutine init
```

Assumed Shaped Arrays (1)

- Assumed shaped arrays allow Fortran subroutines and functions to receive multi-dimensional arrays *without their bounds*;
- Use `lbound()` and `ubound()` to obtain array bounds and use the `contiguous` attribute:

```
subroutine sub1( vec )
  integer :: i
  real, dimension(:), contiguous, intent(out) :: vec
  do i = lbound( vec, 1 ), ubound( vec, 1 )
    ! operate on vec(i)
  end do
end subroutine sub1
```

nag®

Assumed Shaped Arrays (2)

- *The first dimension is defaulted to 1 and if it is another number, it must be specified, e.g.:*

```
real, dimension(0:), contiguous, intent(out) :: vec
```

- The `contiguous` keyword (Fortran 2008) tells the compiler that the array has unit stride, thus elements are contiguous in memory *which helps the compiler to vectorise your code*. In addition, it avoids expensive copying;
- Assumed shaped arrays make subroutine and function calls cleaner and aid better software engineering;
- Assumed shaped arrays (Fortran 90) is a major improvement and shows the strength of the Fortran language and its management of arrays.

Automatic Arrays

- The automatic array feature allows creation of arrays in subroutines:

```
subroutine sub1( vec )
  real, dimension(:), intent(in) :: vec
  real, dimension(size( vec )) :: temp
end subroutine sub1
```

- When the subroutine `sub1` completes the `temp` array is discarded along with all other local variables as they are allocated on the stack;
- If allocating large amounts of memory locally in a function or subroutine, increase the stack size in the Linux shell:

```
ulimit -s unlimited
```

Fortran Pointers (1)

- Fortran 95 introduced pointers. Fortran 77 emulated pointers using Cray pointers. A pointer is an object that points to another variable which is stored in another memory location;
- Pointers can have the following states: *undefined*, *not associated* or *associated*;
- Always assign it to null, so it is in a known state:

```
type(molecule_t), pointer :: m1
m1 => null( )
m1 => molecules(n)
nullify( m1 )
```

nag®

Fortran Pointers (2)

- If a pointer will be pointing to a variable, make sure it has the `target` attribute:

```
real, dimension(N), target :: vec
real, dimension(:), pointer :: vec_p
vec_p => vec
```

- This helps the compiler optimize operations on variables that have the `target` attribute;
- A dangling pointer points to a memory reference which has been deallocated. This causes undefined behaviour! The NAG Fortran Compiler can detect dangling pointers;
- Avoid declaring arrays as pointers as compilers have difficulties vectorising and optimizing operations on them.

Fortran Pointers (3)

- Use the `associated` intrinsic function to check if pointers are associated with a target:

```
if ( associated( x_p )) then
   ! [ ... ]
end if
if ( associated( x_p, x )) then ! if x_p points to x
   ! [ ... ]
end if
```

nag

Fortran Pointers (4)

- Question: what will happen in this case?

```
integer, pointer :: p1
if ( associated( p1 )) then
  print *, ‘p1 is associated’
else
  print *, ‘p1 is not associated’
end if
```

Fortran Pointers (5)

- Pointer `p1` is undefined, thus the code is invalid Fortran. Pointer should be set to `null()` so it is in a *defined* state;
- Compilers will arbitrarily set `p1` to associated or not associated;
- NAG compiler can catch this bug with the `-C=pointer` flag during runtime:

```
Runtime Error: pointy_test.f90, line 7: Undefined pointer
P1 used as argument to intrinsic function ASSOCIATED
Program terminated by fatal error
```

Allocatable Length Strings

- Fortran 2003 now provides allocatable length strings

```
character(len=:), allocatable :: str
str = 'hello'
str = 'hello world' ! string length increases
```

- However, arrays of strings are different:

```
character(len=:), allocatable :: array(:)
allocate( character(len=100) :: array(20) )
```

- To adjust, you must `allocate` and `deallocate`.

Fortran Pre-Processing (1)

- The pre-processor is a text processing tool which is usually integrated into the compiler;
- It is a separate stage and occurs prior to compilation:

```
#ifdef DEBUG
  print *, 'count is', counter
#endif
```

- To assign the macro `DEBUG`, compile with:

```
$ nagfor -c -DDEBUG code.F90
```

Fortran Pre-Processing (1)

- Preprocessing is sometimes used to compile code for different operating systems, e.g. Linux and Windows;
- It is also used to build debug versions of the code which includes printing the status of variables.

Fortran File Extensions (1)

- Modern Fortran codes should either use the `f90` or `F90` file extensions, e.g. `solver_mod.F90` and this is for all modern Fortran standards;
- Files ending with `F90` are pre-processed before being compiled;
- Files ending with `f90` are not pre-processed. It is simply compiled;
- Pre-processor takes a code, processes it, and outputs another code which is then compiled;

Fortran File Extensions (2)

- The `.f90` file extension usually assumes the latest Fortran standard, namely 2008. This can be adjusted with compiler flags;
- Other file extensions are also accepted: `.f95`, `.f03` and `.f08`. The pre-processed versions are `.F95`, `.F03` and `.F08`, respectively.

Fortran Preprocessing using Fypp

- Fypp [1] is a preprocessor and has meta-programming features designed for Fortran;
- Meta-programming involves using the Fortran code with Fypp constructs as input and producing actual Fortran code;
- It has much more powerful features than standard preprocessors;
- Fypp is written in Python and Fypp constructs can include Python expressions;
- Supports iterations, multiline macros and continuation lines;
- Note that preprocessing is not part of the language standard.

[1] https://github.com/aradi/fypp

Fypp Syntax - Control Directives

- Line form:

```
#:if DEBUG > 0
  print *, ‘debugging information. alpha = ’, alpha
#:endif
```

- Inline form:

```
#{if DEBUG > 0}# print *, ‘opt = ’, opt #{endif}#
```

Fypp Syntax - Evaluation Directives

- Line form - can add Python expressions:

```
$:time.strftime('%Y-%m-%d')
```

- Inline form:

```
print *, "Compile date: ${time.strftime('%Y-%m-%d')}$"
character(len=*), parameter :: user = &
                   "${os.environ['USER']}$"
```

Fypp Syntax - Direct Call Directives

- Line form:

```
@:mymacro( a < b )
```

- Inline form:

```
print *, “test result = ”, @{mymacro( a < b)}@
```

Fypp Examples (1)

```
#:if DEBUG > 0
  print *, “debug information. alpha = ”, alpha
#:endif

#:if defined (‘WITH_MPI’)
  use mpi_f08
#:elif defined (‘WITH_OPENMP’)
  use omp_lib
#:else
  use serial
#:end if
```

nag®

Fypp Examples (2)

```
interface myfunc
#:for dtype in ['real', 'dreal', 'complex', 'dcomplex']
  module procedure myfunc_${dtype}$
#:endfor
end interface myfunc

logical, parameter :: hasMpi =#{if defined('MPI')}# .true.
#{else}# .false. #{endif}#

character(len=*), parameter :: comp_date = &
     "${time.strftime('%Y-%m-%d')}$"
```

nag

Fypp Examples (3)

- Line continuation:

```
#:if var1 > var2 &
    & or var2 > var4
  print *, "Doing something here"
#:endif
```

- Creating variables:

```
#:set LOGLEVEL = 2
print *, "LOGLEVEL: ${LOGLEVEL}$"
```

Defining Macros in Fypp

```
#:def assertTrue(cond)
#:if DEBUG > 0
if ( .not. ${cond}$ ) then
  print *, "Assert failed in file ${_FILE_}$, line
${_LINE_}$"
  error stop
end if
#:endif
#:enddef assertTrue
! Invoked via direct call
@:assertTrue( size(myArray) > 0 )
```

Invoking Fypp

- To install Fypp, use:

```
$ pip install fypp
```

- To invoke Fypp, use:

```
$ fypp -m os -m time -DDEBUG=2 code.F90 > code.f90
```

- The `-m` flags are required for additional Python modules;
- The `-D` flag is used to set macros, e.g. `DEBUG` macro is set to 2;
- The above command can be used in a Makefile.

Numerical Kind Types (1)

- For single and double precision data types, use:

```
use, intrinsic :: iso_fortran_env
integer, parameter :: SP = REAL32
integer, parameter :: DP = REAL64
integer, parameter :: QP = REAL128

real(kind=DP) :: alpha, gamma
alpha = 2.33_DP       ! must postfix with _DP
gamma = 1.45E-10_DP ! otherwise value will be _SP
```

- Likewise for `INT8`, `INT16`, `INT32` and `INT64`

Numerical Kind Types (2)

- Printing `alpha = 1.1` and `alpha = 1.1_REAL64` prints:

```
01 1.1000000238418579 1.1000000000000001
```

- Printing `beta = alpha**2` with single and double precision gives:

```
02 1.2100000381469726E+00 1.2100000000000002E+00
```

- The relative error between single and double precision is:

```
03 3.155827165774653E-08
```

Numerical Kind Types (3)

- Unfortunately, GNU Fortran implements `REAL128` as 80 bits (the old Intel extended precision);
- To fully ensure portability, use the following kind constants:

```
integer, parameter :: SP = &
        selected_real_kind( p = 6, r = 37 )
integer, parameter :: DP = &
        selected_real_kind( p = 15, r = 307 )
integer, parameter :: QP = &
        selected_real_kind( p = 33, r = 4931 )
```

- The above constants forces the required precision (`p` decimal places) and range (`r` where $-10^r < \text{value} < 10^r$). The above use the IEEE-754 standard.

nag®

Mixed Mode Arithmetic

- The following automatic type conversions occur in Fortran:

  `integer * real -> real` left hand side must be real

  `integer / real -> real`

  `integer + or - real -> real`

  `real * double -> double` left hand side must be double

  `integer / integer -> integer` but truncation occurs!

  `integer**(-n)` will always be zero for `n > 0`

Precision Bugs (1)

- The following code segments have bugs:

```
real :: a, geom, v, g_p
a = geom * v ** (2/3) ! calculate surface area
g_p = 6.70711E-52

real(kind=REAL64) :: theta
real :: x
x = 100.0_REAL64 * cos( theta ) ! mixing of precisions
```

Precision Bugs (2)

```
real(kind=REAL64) :: d
real :: x, y
d = sqrt( x**2 + y**2 )
```

- Compilers are generally not good at spotting precision bugs;
- To avoid precision bugs, you can use the unify precision feature of the NAG Fortran Compiler.

Type Conversions

- Use the following intrinsic functions when converting between types:

```
int( arg_real, [kind] )
real( arg_int, [kind] )
```

- Use the generic functions for all types:

Generic Name (modern)	Specific Name (old)	Argument Type
sqrt	csqrt	complex
sqrt	dsqrt	double precision
sqrt	sqrt	real

Fortran Module (1)

- *Modules allow type checking for function/subroutine arguments at compile time so errors are quickly identified*;
- Fortran module files are pre-compiled header files which means codes compile faster than comparable C/C++ codes;
- However, they must be re-create for different compilers and sometimes for the same compiler but different versions;

Fortran Modules (2)

```
module module_mod
   use anotherModule_mod
   implicit none

   private :: ! list private symbols
   public :: ! list public symbols
   ! define variables, constants and types
   real, protected :: counter = 0
contains
   ! define functions and subroutines here
end module module_mod
```

Fortran Module Names

- When naming internal modules, use the `mod` suffix so the name does not clash with another symbol:

```
module matrix_mod
    ! [ … ]
end module matrix_mod
```

- Put the above module in a file called `matrix_mod.F90` so it is clear that it contains the named module only. **Only put one module per file**;
- **Always lower case the filename containing a module**. This helps pattern matching in GNU makefile.

Using Modules

- To use a module in your code:

```
use module_mod
```

- To use a subset of the procedures from a module:

```
use module_mod, only : Solve_system, Init_system
```

- To rename an entity in a module:

```
use module_mod, Solve_system => Solve_linear_system
```

- The rename feature might be required to avoid a name clash;
- You can use both:

```
use module_mod, only : Solve_system => Solve_linear_system
```

Basic Polymorphism in Modules

```
module vector_mod
  interface my_sum
    module procedure real_sum
    module procedure int_sum
  end interface
contains
  function real_sum( vec )
    real, intent(in) :: vec(:)
  end function real_sum
  function int_sum( vec )
    integer, intent(in) :: vec(:)
  end function int_sum
end module vector_mod
```

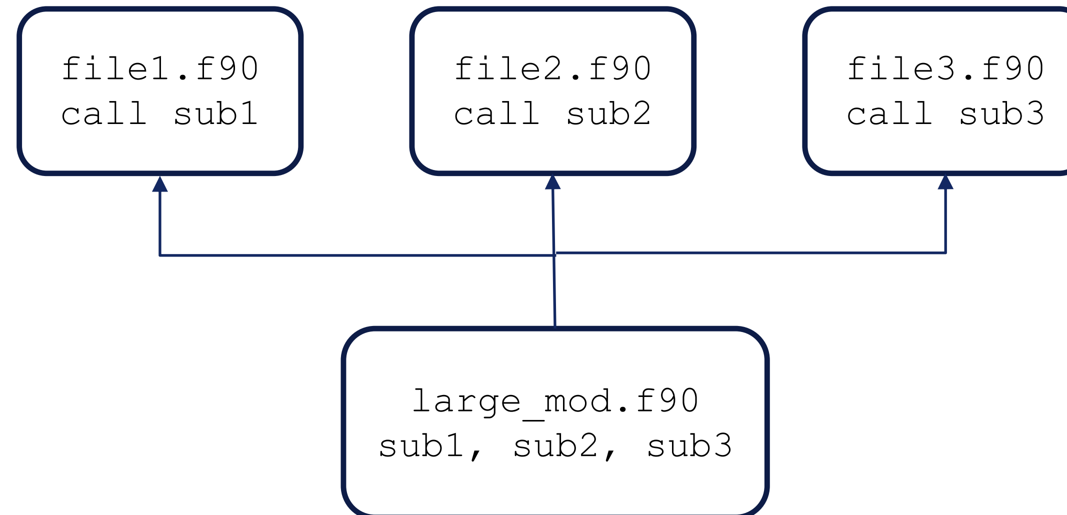
```
program main_prog
  use vector_mod

  implicit none
  integer :: veci = [ 1, 2, 3 ]
  real :: vecr = [ 1.0, 2.0, 3.0 ]

  print *, my_sum( vecr )
  print *, my_sum( veci )
end program main_prog
```

Fortran Submodules (1)

- Fortran 2008 introduced the submodule feature which allows the separation of a) function, subroutine and variable *declarations* (Fortran interfaces) and b) function and subroutine *implementations*;
- Submodules subsequently speed up the build process in addition minimising the number of files that are affected during a change;
- A module is created which includes variable declarations and function/subroutine interfaces. Interfaces are declarations of the functions/subroutines;
- A submodule contains the implementations of functions and subroutines;

Fortran Submodules (2)

- Current situation: `file1.f90`, `file2.f90` and `file3.f90` all use `large_mod` and call `sub1( )`, `sub2( )` and `sub3( )`, respectively;



- A change in `sub3` (in `large_mod.f90`) will trigger the rebuild of all files (`file1.f90`, `file2.f90` and `file3.f90`) which is obviously unnecessary;

Fortran Submodules (3)

- In addition, separating into two files reduces the risk of bugs being introduced - further increasing software abstraction;
- To use the submodule feature, function and subroutine interfaces must not change. Interfaces very rarely change - it is the implementation that changes more often;
- Fortran submodules are supported by the Intel compiler version 16.0.1 and GNU Fortran 6.0;

Fortran Submodules (4)

- Firstly, define the module (in file `large_mod.f90`):

```
module large_mod
  public :: sub1, sub2, sub3
interface
  module subroutine sub1( a )
    real, intent(inout) :: a
  end subroutine sub1
  ! same for sub2( ) and sub3( )
end interface
end large_mod
```

- The above module is comparable to a C/C++ header file;

nag

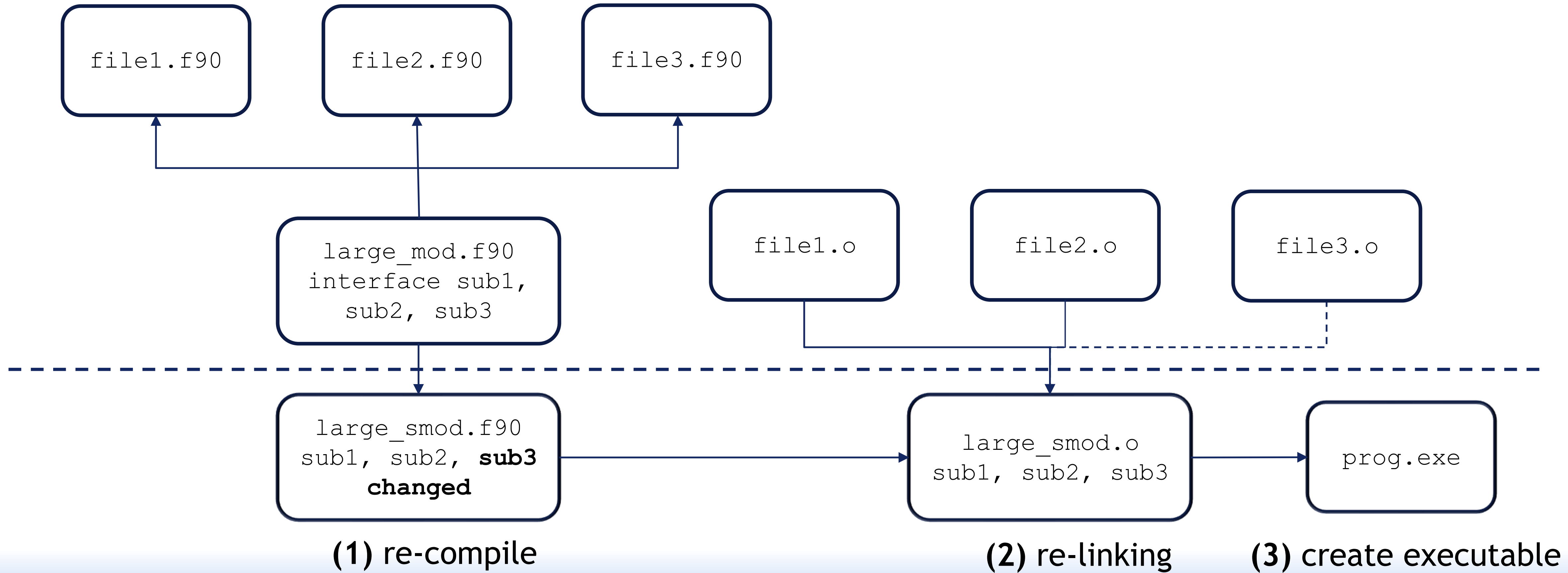
Fortran Submodules (5)

- Secondly, define the submodule (in file `large_smod.f90`) with `sub1( )`:

```
submodule (large_mod) large_smod
contains
  module subroutine sub1( a )
    real, intent(inout) :: a

    a = a**2
  end subroutine sub1 ! define sub2( ) and sub3( )
end submodule large_smod
```

- Compiling the above submodule creates a file `large_mod@large_smod.smod` (or `module@submodule.smod`)

Fortran Submodules (6)



Fortran Loops (1)

- Always use DO loops with fixed bounds (trip counts) *without* `cycle` or `exit` statements if possible:

```
do i = 1, N
   ! some code
end do
```

- There is more chance the compiler can optimize (e.g. vectorise) the above loop. Such loops can also be parallelised using OpenMP;
- Use the loop counter as an index for arrays (`i` in the above example);
- Avoid branching in loops as this prevents compiler optimizations;

Fortran Loops (2)

- While loop structure:

```
do while ( logical-expression )
  ! [ .... ]
end do
```

- Avoid `do while` loops. If you are, parallelise the block of code within the loop;
- While loops are sometimes required, e.g. for iterative algorithms that continue until a solution (within error bounds) is achieved;

Do Concurrent Loops (1)

- Fortran `forall` has been obsoleted in the 2018 standard due to performance issues (implicit barrier after each statement);
- The `do concurrent` construct has replaced `forall` loops:

```
do concurrent ( i = 1:100 )
   vec(i) = vec1(i) + vec2(i)
end do
```

- **All iterations are completely independent.** The compiler is likely to vectorise the above;
- The `exit`, `stop` and `cycle` statements are not permitted and no branching outside of it is allowed;

Do Concurrent Loops (2)

- Can also include masking:

```
do concurrent ( i = 1:n, j = 1:m, &
                i /= j .and. A(i, j) > 1.0 )
  C(i, j) = log( A(i, j) )
end do
```

- **Fortran 2018**:

```
do concurrent ( integer(INT64) :: i = 1:n, j = 1:m, &
                i /= j .and. A(i, j) > 1.0 )
  C(i, j) = log( A(i, j) )
end do
```

Do Concurrent Loops (3)

- **Fortran 2018** clauses scope the variables in the loop with the aim of improving performance, i.e. not serialising the loop:

```
real :: a(10), x
do concurrent ( i = 1:10 ) local (x) shared(a, b)
  if ( a(i) > 0.0 ) then
    x = sqrt( a(i) )
    a(i) = a(i) - x
  else
    x = a(i)**2
    a(i) = a(i) - x
  end if
end do
```

nag

IEEE Floating Point Arithmetic

- Operating on floating point data can raise exceptions that can indicate an abnormal operation, as defined in the IEEE-754 standard;
- The exception that be raised as defined by IEEE-754 are:

IEEE Exception (Flag)	Description	Default Behaviour
IEEE_DIVIDE_BY_ZERO	Division by zero	Signed ∞
IEEE_INEXACT	Number is not exactly represented	Rounded to nearest, overflow or underflow
IEEE_INVALID	Invalid operation such as √-1, operation involving ∞, NaN operand	Quiet NaN (not a number)
IEEE_OVERFLOW	Rounded result larger in magnitude than largest representable format	+∞ or -∞
IEEE_UNDERFLOW	Rounded result smaller than smallest representable format	Subnormal or flushed to zero

nag®

IEEE Compiler and System Support

- Floating point exceptions are usually handled by the compiler, but they are not standard;
- The Fortran 2003 provides an API to manage exceptions;
- To determine what exceptions are supported:

```
use ieee_arithmetic
ieee_support_datatype( 1.0_REAL32 ) ! for single
ieee_support_datatype( 1.0_REAL64 ) ! for double
ieee_support_datatype( 1.0_REAL128) ! for quad
```

- The above will return Boolean `.true.` or `.false.`

nag®

IEEE Exception Support

- To determine what exceptions are support for your data type and compiler/system (returns `.true.` or `.false.`):

```
ieee_support_flag( ieee_all(i), 1.0_PREC )
```

where

```
ieee_all(1) = 'IEEE_DEVIDE_BY_ZERO'
ieee_all(2) = 'IEEE_INEXACT'
ieee_all(3) = 'IEEE_INVALID'
ieee_all(4) = 'IEEE_OVERFLOW'
ieee_all(5) = 'IEEE_UNDERFLOW'
```

PREC = precision which either `REAL32`, `REAL64` or `REAL128`.

IEEE Exceptions (1)

- Exception handling is done via subroutines and is called immediately after an operation:

```
x = … ! floating point operation
call ieee_get_flag( ieee_flag, exception_occurred )
```

where

ieee_flag = IEEE_OVERFLOW, IEEE_UNDERFLOW, IEEE_INEXACT, IEEE_DEVIDE_BY_ZERO, IEEE_INVALID

exception_occurred = returns logical .true. or .false. depending on whether the exception occurred

IEEE Exceptions (2)

- To determine if floating point variable is a NaN (not a number), use:

  ```
  ieee_is_nan( x )
  ```

  which returns logical `.true.` or `.false.`
- To determine if a floating point variable is finite or infinite, use:

  ```
  ieee_is_finite( x )
  ```

  which returns logical `.true.` or `.false.`
- For rounding modes, use:

  ```
  call ieee_get_rounding_mode( value )
  call ieee_set_rounding_mode( value )
  ```

  where `value` is `type(ieee_round_type)` which can be one of
  `ieee_nearest, ieee_to_zero, ieee_up, ieee_down`

IEEE Exceptions Testing

- Testing for IEEE exceptions after every numeric computation will completely slow down calculations;
- Check for IEEE exceptions after important calculations;
- Prefix the check with a macro which is enabled when testing:

```
x = … ! floating point operation
#ifdef DEBUG
call ieee_get_flag( IEEE_OVERFLOW, exception_occurred )
#end if
```

- The `-ieee=stop` NAG compiler flag will terminate execution of the code on floating point overflow, division by zero or invalid operand.

Good API Characteristics

- It provides a high level description of the behaviour of the implementation, abstracting the implementation into a set of subroutines, encapsulating data and functionality;
- Provides the building blocks of an application;
- They have a very long life, so design your API carefully. A change in the API will require a change in codes that use the API;
- They are developed independently of application code and can be used by multiple applications of *different languages*;
- *The API should be easy to use and difficult to misuse*. Always use the Fortran `intent` keyword.

API Design

- If a function/subroutine has a long list of arguments, encapsulate them in a user defined data type:

```
type square_t
   real :: x1, y1, x2, y2
end type square_t
subroutine area( sq1 )
   type(square_t) :: sq1
end subroutine area
```

- Use the `contiguous` (unit stride) attribute for assumed shaped arrays which will allow the compiler to optimize code.

Optional Arguments

- Use optional arguments to prevent code duplication:

```
subroutine Solve_system( A, b, x, rtol, max_iter )
  real, dimension(:,:), intent(in) :: A
  real, dimension(:), intent(inout) :: x,
  real, dimension(:), intent(in) :: b
  real, intent(in), optional :: rtol, max_iter

  if ( present( rtol )) then

  end if
end subroutine Solve_system

call Solve_system( A, b, x, rtol = e, max_iter = n )
```

nag®

Using Optional Arguments Carefully (1)

```
subroutine log_entry( message, header )
  character(len=*), intent(in) :: message
  logical, optional, intent(in) :: header
  ! incorrect. can you see why?
  if ( present( header ) .and. header ) then
    print *, 'This is the header'
  end if
  print *, message
end subroutine log_entry
```

nag®

Using Optional Arguments Carefully (2)

```
subroutine log_entry( message, header )
  character(len=*), intent(in) :: message
  logical, optional, intent(in) :: header
  ! correct way of doing it
  if ( present( header )) then
    if ( header ) then
      print *, 'This is the header'
    end if
  end if
  print *, message
end subroutine log_entry
```

nag

Fortran Functions

- You can the `result` clause when defining functions:

```
function delta( a, b ) result ( d )
   real, intent(in) :: a, b
   real :: d ! intent not required.
             ! it is defaulted to intent(out)
   d = abs( a - b )
end function delta
```

Procedure Variables

- It is recommended to list dummy arguments first followed by local variables, e.g.

```
subroutine swap( a, b )
  integer, intent(inout) :: a, b ! dummy arguments first
  integer :: temp                ! local variables after

  temp = a; a = b; b = temp
end subroutine swap
```

Pure Subroutines and Functions

- Subroutines and functions can change arguments through the `intent` feature but this can be unsafe for multi-threaded code;
- When subroutines change arguments, this is known to create *side effects* which inhibit parallelisation and/or optimization;
- *Declare your function as pure which tells the compiler that the function does not have any side effects*:

```
pure function delta( a, b ) result( d )
  real, intent(in) :: a, b
  real :: d

  d = a**2 + b
end function
```

Elemental Subroutines and Functions

- Elemental subroutines with scalar arguments are applied to arrays and must have the same properties as pure subroutines, i.e. no side effects;
- This allows compilers to vectorise operations on arrays:

```
elemental function sqr( x, s ) result( y )
!$omp declare simd(sqr) uniform(s) linear(ref(x))
   real, intent(in) :: x, s
   real :: y
   y = s*x**2
end function sqr

print *, sqr( [ 1.0, 2.0, 3.0 ], 2.0 ) ! print 2.0, 8.0, 18.0
```

- Use the `-qopenmp-simd` Intel compiler flag to vectorise the above code.

Debug Mode

- When developing libraries, have a debug option that prints additional information for debugging:

```
if ( debug ) then
  print *, 'value of solver option is = ', solver_option
end if
```

- This will not slow your code down as this will be removed using the compiler's dead code elimination optimization (`debug = .false.`);
- Do not let your library exit the program - return any errors using an integer error flag;
- Zero for success and non-zero for failure. Non-zero value will depend on type of failure, e.g. 1 for out of memory, 2 for erroneous parameter, 3 for file not found, etc.

Library Symbol Namespace

- When developing a library, ensure subroutines, functions and constants are all prefixed with the name of the library;
- For example, when creating a library called HAWK:

```
use HAWK
call HAWK_Init( ierr )
n = HAWK_MAX_OBJECTS
call HAWK_Finalize( ierr )
```

- This way, you are not “polluting” the namespace;
- Users know where the subroutine and constants are from.

Deleted and Obsolescent

- Would be better not to know about these statements at all;
- Mostly important for legacy (~ 30+ years old) code developers;
- Very few statements/features have been deleted/made obsolete;
- Tabulated for convenience;
- Obsoleted means a standard has been labelled for deletion and there is already a modern structure;
- Deleted means a standard has been removed from the language;
- It is likely that some compilers still support deleted features.

Deleted Features

	OBS	DEL
Real and double precision DO variables	90	95
Branching to an END IF statement from outside its block	90	95
PAUSE statement	90	95
ASSIGN and assigned GO TO statements and assigned FORMAT specifiers	90	95
H edit descriptor	90	95
Arithmetic IF	90	18
Shared DO termination and termination on a statement other than END DO or CONTINUE	90	18

Real and double precision DO variables

- Deleted

```
do x = 0.1, 0.8, 0.2
  ...
  print *, x
  ...
end do
```

- Alternative

```
do x = 1, 8, 2
  ...
  print *, real(x)/10.0
  ...
end do
```

- Use integers

Branching to an END IF statement from outside its block

- **DISCLAIMER:**
  try to avoid GO TOs

- Deleted

```
    go to 100
    ...
    if (scalar-logical-expr) then
      ...
100 end if
```

- Alternative

```
    go to 100
    ...
    if (scalar-logical-expr) then
      ...
    end if
100 continue
```

- Branch to the statement following the END IF statement or insert a CONTINUE statement immediately after the END IF statement

PAUSE statement

- Suspends execution

- Deleted

```
pause [stop-code]
```

- Alternative

```
write (*,*) [stop-code]
read (*,*)
```

H edit descriptor

- Hollerith edit descriptor

- Deleted

```
print "(12Hprinted text)"
```

- Alternative

```
print "('printed text')"
```

- Use characters

Arithmetic IF

- IF (*scalar-numeric-expr*) rather than IF (*scalar-logical-expr*)

- Deleted

```
    if (x) 100, 200, 300
100 continue !x negative
    block 100
200 continue !x zero
    block 200
300 continue !x positive
    block 300
```

- Alternative

```
if (x < 0) then
  block 100
  block 200
  block 300
else if (x > 0) then
  block 300
else
  block 200
  block 300
end if
```

- Use IF or SELECT CASE construct or IF statement

Shared DO termination and termination on a statement other than END DO or CONTINUE

- Deleted

```
    do 100 i = 1, n
      ...
      do 100 j = 1, m
        ...
100     k = k + i + j
```

- Alternative

```
do i = 1, n
  ...
  do j = 1, m
    ...
    k = k + i + j
  end do
end do
```

• Use END DO

DO Loops

- Obsolescent

```
DO 100 i = 1, 100
a(i) = REAL( i )
100 b(i) = 2. * c(i)
! or
DO 200 i = 1, 100
  a(i) = REAL( i )
  b(i) = 2. * c(i)
200 continue
```

- Alternative

```
do i = 1, 100
  a(i) = real( i )
  b(i) = 2.0 * c(i)
end do
```

Obsolescent Features

	OBS	DEL
Alternate return	90+	-
Computed GO TO statement	95+	-
Statement functions	95+	-
DATA statements amongst executable statements	95+	-
Assumed length character functions	95+	-
Fixed form source	95+	-
CHARACTER* form of CHARACTER declaration	95+	-
ENTRY statements	08+	-
Label form of DO statement	18+	-
COMMON and EQUIVALENCE statements and BLOCK DATA program unit	18+	-
Specific names for intrinsic functions	18+	-
FORALL construct and statement	18+	-

Alternate return

- Obsolescent

```
    call sub (x, *100, *200, y)
    block A
100 continue
    block 100
200 continue
    block 200
```

```
subroutine sub (a, *, *, b)
  ...
  return 2
  ...
end subroutine sub
```

- Use integer return with IF or SELECT CASE construct

- Alternative

```
call sub(x, r, y)
select case (r)
  case (1)
    block 100
    block 200
  case (2)
    block 200
  case default
    block A
    block 100
    block 200
end select
```

```
subroutine sub (a, s, b)
  ...
  s = 2
  ...
end subroutine sub
```

Computed GO TO statement

- Obsolescent

```
    go to (100, 200) x
    block A
100 continue
    block 100
200 continue
    block 200
```

- Alternative

```
select case (x)
  case (1)
    block 100
    block 200
  case (2)
    block 200
  case default
    block A
    block 100
    block 200
  end select
```

- Use SELECT CASE (preferable) or IF construct

Statement functions

- Obsolescent

```
real :: axpy, a, x, y
...
axpy (a, x, y) = a*x+y
...
mad = axpy (p, s, t)
...
```

- Alternative

```
mad = axpy (p, s, t)
...
contains
  real function axpy (a, x, y) result (r)
    implicit none
    real, intent (in) :: a, x, y
      r = a*x+y
  end function axpy
```

  - Use internal function

CHARACTER* form of CHARACTER declaration

- Obsolescent

```
character*11 :: x
```

- Alternative

```
character([len=]11) :: x
```

nag

Common Blocks

- Obsolescent

```
PROGRAM COMMON_STATEMENT
  integer i
  real r
  COMMON / comm1 / i, r
  call sub1
contains
  SUBROUTINE sub1
    integer i1
    real r1
    COMMON / comm1 / i1, r1
  END SUBROUTINE sub1
END PROGRAM
```

- Alternative

```
program module_statement
  use comm1_mod
  call sub1
contains
  subroutine sub1
    use comm1_mod
  end subroutine sub1
end program

module comm1_mod
  integer :: i
  real :: r
end module comm1_mod
```

PARAMETER Statements

- Obsolescent

```
INTEGER SIZE
PARAMETER ( SIZE = 100 )
! or
DATA SIZE /100/
```

- Alternative

```
integer, parameter :: size = 100
```

EQUIVALENCE Statements

- Obsolescent

```
REAL r1(10), r2(10)
EQUIVALENCE ( r1, r2 )
```

- Alternative

```
real, target :: r1(10)
real, pointer :: r2(:)

r2 => r1
```

nag®

NAG Fortran Compilation Messages (1)

- **Info** - informational message, highlighting an aspect of the code in which the user may be interested in;
- **Warning** - the source appears to have an error and worth investigating;
- **Questionable** - some questionable aspect has been found in the code;
- **Extension** - some non-standard conforming code has been detected;
- **Obsolescent** - obsoleted feature has been used in the code. It is recommended to replace it with a more modern feature;
- **Deleted** - a deleted feature has been used. You should definitely replace it with a more modern feature;

NAG Fortran Compilation Messages (2)

- The compiler can be used to detect obsoleted and deleted Fortran features and modernisation efforts can be subsequently made;
- Language extensions can also be replaced with standard Fortran for the purpose of portability;
- Questionable and warning messages can also be used for error detection;
- **Diagnostic messages are much more comprehensive than other compilers**;
- Runtime checks can also be carried out by the NAG compiler - to be presented.

NAG Fortran Compiler Polish (1)

- The NAG compiler has some refactoring features;

  ```
  $ nagfor =polish [options] code.f90 -o code.f90_polished
  ```

  where the `options` can be one of:

- `-alter_comments` - enable options to alter comments;
- `-array_constructor_brackets=X` - specify the form to use for array constructor delimiters, where `X` is one of `{Asis,Square,ParenSlash}`;
- `-idcase=X` and `-kwcase=X` - set the case to use for identifiers (variables) and keywords. `X` must be `{C,L,U}`;
- `-margin=N` - set the left margin (initial indent) to `N` (usually 0);

NAG Fortran Compiler Polish (2)

- `-indent=N` - indent statements within a construct by `N` spaces from the current indentation level;
- `-indent_comment_marker` - when indenting comments, the comment character should be indented to the indentation level;
- `-indent_comments` - indent comments;
- `-indent_continuation=N` - indent continuation lines by an additional `N` spaces;
- `-kind_keyword=X` - specifies how to handle the `KIND=` specifier in declarations. X must be one of `{Asis,Insert,Remove}`, e.g. `real(REAL64) :: alpha` becomes `real(KIND=REAL64) :: alpha`

NAG Fortran Compiler Polish (3)

- `-relational=X` - specifies the form to use for relational operators, X must be either `F77` – (use `.EQ.`, `.LE.`, etc.) or `F90+` (use `==`, `<=`, etc.)
- `-dcolon_in_decls=Insert` - add double colons after variable declarations, e.g. `integer i` becomes `integer :: i`
- `-character_decl=Keywords` - change old-style character declarations to new-style, e.g. `character*11 :: str` to `character(len=11) :: str`

NAG Fortran Compiler Polish Example (1)

```
PROGRAM OLD_FORTRAN
  IMPLICIT NONE
  INTEGER i
  REAL VEC(100), RESULT(100)

  DO 100 I = 1, 100
  VEC(I) = 1.0
  RESULT(I) = VEC(I)**2
  100 CONTINUE
END PROGRAM OLD_FORTRAN
```

```
program new_fortran
  implicit none
  integer :: i
  real :: vec(100), result(100)

  do i = 1, 100
    vec(i) = 1.0
    result(i) = vec(i)**2
  end do
end program new_fortran
```

NAG Fortran Compiler Polish Example (2)

- Command used in previous example is:

```
$ nagfor =polish -margin=0 -indent=2 -kwcase=L -idcase=L \
                 -dcolon_in_decls=Insert old_fortran.f90 \
                 -o new_fortran.f90
```

Enhanced Polisher

- Can convert fixed format (FORTRAN 77) to free format modern Fortran;
- Code cannot have any compilation errors;
- Can add keywords to actual arguments in procedure references;
- Compiler command:

```
$ nagfor =epolish src.f -o src.f90
```

- Creates modern Fortran code `src.f90` from the legacy FORTRAN 77 code `src.f`

Enhanced Polisher Example

```
      DOUBLE PRECISION DX(*)
*     .. Local Scalars ..
      DOUBLE PRECISION DTEMP
      INTEGER I,M,MP1,NINCX
*     .. Intrinsic Functions ..
      INTRINSIC DABS,MOD
      DASUM = 0.0d0
      DTEMP = 0.0d0
      IF (N.LE.0 .OR. INCX.LE.0)
        RETURN
      IF (INCX.EQ.1) THEN
```

Fixed form FORTRAN 77

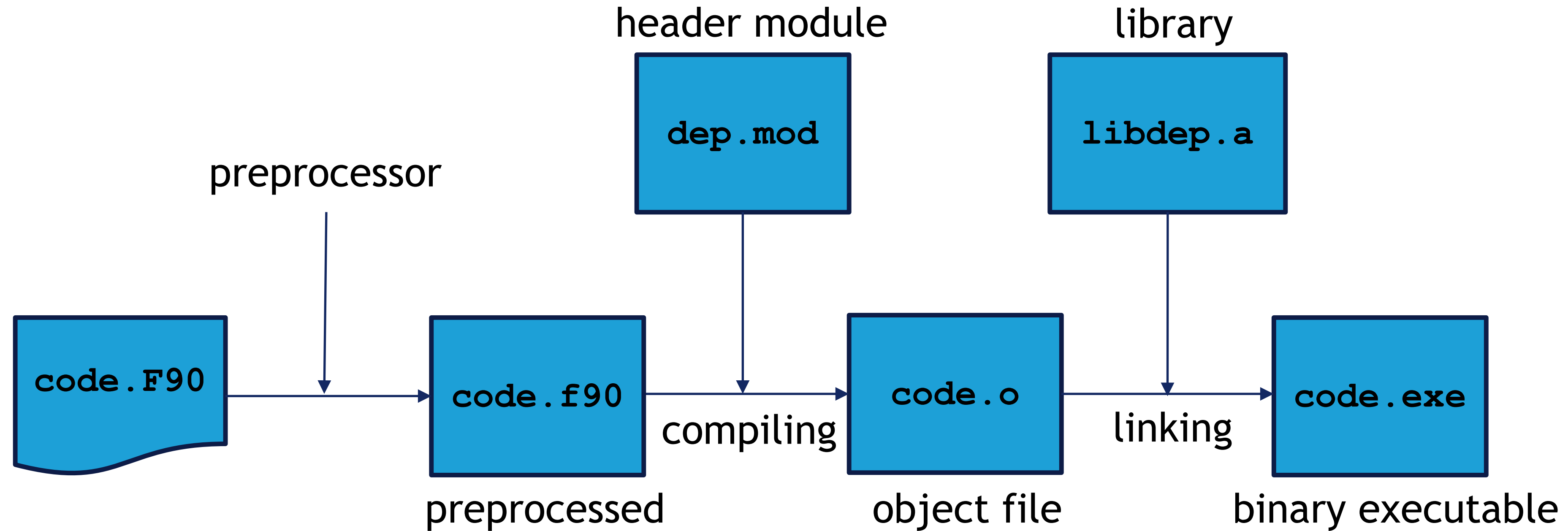
```
  Double Precision :: dx(*)
!       .. Local Scalars ..
  Double Precision :: dtemp
  Integer :: i, m, mp1, nincx
!       .. Intrinsic Functions ..
  Intrinsic :: dabs, mod
  dasum = 0.0D0
  dtemp = 0.0D0
  If (n<=0 .Or. incx<=0) Return
  If (incx==1) Then
```

Free form modern Fortran

Compiling Fortran Modules

- When compiling the file `matrix_mod.F90` the compiler creates two files;
- The first file is `matrix_mod.mod` which is the Fortran header module file. Notice that the filename is in lowercase and *this file does not contain any subroutine or function symbols*. This header module file is required for **compilation only**;
- The second file is `matrix_mod.o` which is the Linux object file *which contains the subroutine and function symbols*. This object file is required for **linking only**.

Building of Codes



Build Commands (1)

- Source code is compiled and header modules (`*.mod`) are *included*:

```
$ nagfor -c -I/path/to/depmod code.F90
```

- The header modules resolve constant *symbols*, e.g. π or e;
- This will create object file `code.o` which needs to be linked to static or shared libraries:

```
$ nagfor code.o -L/path/to/libdep -ldep -o code.exe
```

which will link `libdep.a` (static) or `libdep.so` (shared). This will *resolve function* or *subroutine symbols.* The Linux linker will default to shared library;

nag®

Build Commands (2)

- Static link will bundle code into final executable whereas shared link will load shared library at run time;
- Path to shared library must be specified via the `LD_LIBRARY_PATH` environment variable and multiple paths are colon separated.
- If both static and shared libraries exist in the same directory, then the Linux linker will select the shared library by default;

Build Commands (3)

- To determine which shared libraries are required:

```
$ ldd workshare.exe
linux-vdso.so.1 =>  (0x00007ffc6ebdf000)
libgfortran.so.3 => /lib64/libgfortran.so.3
(0x00002b046d5d2000)
libm.so.6 => /lib64/libm.so.6 (0x00002b046d8fa000)
```

- Statically linking reduces the time the executable code gets loaded into memory. Subsequently, static libraries do not need to exist on the target system;
- *For performance at large number MPI of ranks, it is recommended to statically link even though your binary executable will become larger;*

Build Commands (4)

- However, static linking will only bundle in procedures (symbols) that are actually being called and not the entire static library;
- When linking with the compiler, *it actually calls the Linux linker* `ld` but it is good practice to use the compiler because it automatically links with the compiler's runtime library.

Ordering Libraries During Linking

- When linking multiple libraries with dependencies, the order of the libraries during linking is crucial;
- Otherwise you will get the dreaded "undefined symbol" errors;

```
$ nagfor code.o -L/usr/lib/netcdf-4.0 -lnetcdff -lnetcdf \
                -o code.exe
```

- The `netcdff` library (Fortran bindings) calls subroutines from the `netcdf` library (C implementation) so *it must be listed in the above order*.

Creating Libraries

- Linking with a large number of object files from Fortran modules can be tedious especially when they need to be correctly ordered;
- Create a single library which contains all object files by using the Linux `ar` command:

```
$ ar rc libfmw.a obj1.o obj2.o obj3.o obj4.o
```

- Prefix the name of library with `lib` followed by name of library (`fwm` in this example) and with the `.a` extension;
- When the main code needs to link with `libfmw.a` use the link flags:

```
$ nagfor main.o -L/path/to/fmw -lfmw -o main.exe
```

File Formats

- Executables, static object files, shared object files and core dumps are stored in the Linux Executable and Linking Format (ELF);
- ELF tools include `nm`, `readelf` and `objdump` which can be used to examine object files for subroutines;
- *Fortran module header (`.mod`) files are compiler specific and will only work with the compiler it was created with.* Sometimes the module header files change between different versions of the same compiler, e.g. GNU Fortran 6.0 and 7.0;
- Therefore, it is always best to recompile from source for compatibility and performance reasons.

NAG Fortran Compiler

- The NAG Fortran Compiler is one of the most comprehensive code checking compilers;
- It checks for possible errors in code and rigorously checks for standards conformance to ensure portability;
- Has unique features which aid good software development;
- Was the first compiler to implement the Fortran 90 standard which was the biggest revision to modernise the language;
- NAG compiler documentation can be found at [1].

[1] https://www.nag.co.uk/nag-compiler

NAG Fortran Compiler Usage

- Usage syntax is:

```
$ nagfor [mode] [options] fortran_source_file.f90
```

where `[mode]` is one of:

`=compiler` - this is the default mode;

`=depend` - analyses module dependencies in specified files;

`=interfaces` - produces a module interface for subroutines in a file;

`=polish` - polishes up the code (already discussed);

`=unifyprecision` - Unify the precision of floating-point and complex entities in Fortran files.

NAG Fortran Compiler Dependency Analyser

- The NAG dependency analyser takes a set of Fortran files and produces module dependency information:

  `$ nagfor =depend -otype=`*`type`* `*.f90`

  where *`type`* is one of:

  `blist` - the filenames as an ordered build list

  `dfile` - the dependencies in Makefile format, written to separate `file.d` files

  `info` - the dependencies as English descriptions

  `make` - the dependencies in Makefile format

NAG Fortran Compiler Interface Generator

- Interfaces can be generated for source files that contain Fortran subroutines. Interfaces allow argument checking at compile time:

```
$ nagfor =interfaces -module=blas_mod *.f
```

- The above will create `blas_mod.f90` which will contain interfaces for all Fortran 77 files in current working directory;
- The output is a Fortran 90 module file which can be included in a Fortran 90 code via the `use blas_mod` statement;
- Remember to include the path to `blas_mod.mod` at compiler time:

```
$ nagfor -I/path/to/blas_mod -c code.f90
```

NAG Fortran Compiler Unify Precision

- This feature unifies the precision in Fortran files to a specified kind parameter in a module:

```
$ nagfor =unifyprecision -pp_name=DP \
     -pp_module=types_mod code.f90 -o code.f90_prs
```

- The above will create file `code.f90_prs` that forces real types to be of kind `DP`, e.g.

```
use types_mod, only : DP
real(kind=DP) :: tol, err
```

NAG Fortran Compiler Code Checking (1)

`-f95`, `-f2003`, `-f2008` - checks the code is Fortran 95, 2003 and 2008 (default) standards compliant, respectively;

`-gline` - this flag will do a subroutine trace call when a runtime error has occurred;

`-mtrace` - trace memory allocation and deallocation. Useful for detecting memory leaks;

`-C=check` - where `check` can be `array` for array out of bounds checking, `dangling` for dangling pointers, `do` for zero trip counts in do loops, `intovf` for integer overflow and `pointer` for pointer references;

NAG Fortran Compiler Code Checking (2)

- For simplicity, use the following flags to do all the checks:

```
$ nagfor -C=all -C=undefined -info -g -gline
```

- The NAG compiler is able to spot 91% of errors [1]:

Run-time Error	Absoft	g95	gfortran	Intel	Lahey	NAG	Pathscale	PGI	Oracle
Percentage Passes [1]	34%	45%	53%	53%	92%	91%	38%	28%	42%
TFFT execution time with diagnostic switches (seconds)[2]	10	16	6	12	446	60		19	9

- The NAG Fortran Compiler can catch errors at either compile time, e.g. non-standard conforming code, or it can catch errors at run time with a helpful error message compared to "segmentation fault".

[1] http://www.fortran.uk/fortran-compiler-comparisons-2015/intellinux-fortran-compiler-diagnostic-capabilities/

Forcheck - Static Analysis Tool

- Forcheck is a static analysis tool which analyses Fortran code without executing them;
- Locates bugs early on in development, potentially saving you a lot of time compared to finding bugs during runtime;
- Much more comprehensive checking than compilers. Some compilers tend to emphasise on performance rather than correctness.

Forcheck Dummy Argument Checking

- Fortran code:

```
subroutine foo( a, b )
  real :: a
  real, optional :: b
  a = b**2 ! not checking to see if b is present
end subroutine foo
```

- Analysis output:

```
(file: arg_test.f90, line: 14)
B
**[610 E] optional dummy argument unconditionally used
```

Forcheck Dummy Argument Intent Checking

- Dummy arguments should always be scoped with the `intent` keyword;
- Command:

```
$ forchk -intent arg_test.f90
```

- Analysis output:

```
B
**[870 I] dummy argument has no INTENT attribute
          (INTENT(IN) could be specified)
```

Forcheck Actual Argument Checking

- Fortran code:

```
call foo( 1.0, b )
```

- Analysis output:

```
        7  call foo( 1.0, b )
(file: arg_test.f90, line: 7)
FOO, dummy argument no    1 (A)
**[602 E] invalid modification: actual argument is constant
or expression
```

Forcheck Precision Checking (1)

- Fortran code:

```
real(kind=REAL64) :: d
real(kind=REAL32) :: s
s = d**2 ! will also be detected by GNU Fortran
d = s**2 ! will not be detected by GNU Fortran
```

- Analysis output - possible truncation:

```
(file: precision.f90, line: 11)
s = d**2
**[345 I] implicit conversion to less accurate type
```

Forcheck Precision Checking (2)

- Analysis output - subtle precision bug:

```
(file: precision.f90, line: 12)
d = s**2
**[698 I] implicit conversion to more accurate type
```

nag

Runtime Checking

- Static analysis checks are easy ways to detect obvious bugs but they are ultimately very conservative. When they say there is a bug, they are correct;
- Static analysis tools are limited in what they can achieve particularly for large codes where there can be variables that are defined in complex IF statements;
- This requires runtime checks to ultimately check for potential bugs with a comprehensive error checking compiler such as the NAG Fortran Compiler;
- The NAG Fortran Compiler also prints helpful error messages to help locate sources of bugs instead of the dreaded “segmentation fault”.

NAG Compiler Optional Argument Detection

- Compile command (if Forcheck cannot detect this):

```
$ nagfor -C=present arg_test.f90 -o arg_test.exe
```

- Fortran code:

```
call foo( a )
subroutine foo( a, b )
  real, intent(out) :: a
  real, intent(in), optional :: b
  a = b**2
end subroutine foo
```

- Helpful runtime error message and not just segmentation fault:

```
Runtime Error: arg_test.f90, line 14: Reference to OPTIONAL argument B which is not PRESENT
```

NAG Compiler Dangling Pointer Detection

- Build command:

```
$ nagfor -C=dangling p_check.f90 -o p_check.exe
```

- Fortran code:

```
real, dimension(:), allocatable, target :: vec
real, dimension(:), pointer :: vec_p

allocate( vec(1:100) )
vec_p => vec; deallocate( vec )
print *, vec_p(:)
```

- Runtime output - NAG compiler is the only Fortran compiler that can check this:

```
Runtime Error: p_check.f90, line 12: Reference to dangling
pointer VEC_P
Target was DEALLOCATEd at line 10 of pointer_check.f90
```

NAG Compiler Undefined Variable Detection

- Compile command:

```
$ nagfor -C=undefined undef_test.f90 -o undef_test.exe
```

- Fortran code:

```
real, dimension(1:11) :: array
array(1:10) = 1.0
print *, array(1:11)
```

Runtime output:

```
Runtime Error: undef_test.f90, line 7: Reference to undefined variable ARRAY(1:11)
Program terminated by fatal error
```

NAG Compiler Procedure Argument Detection

- Compile command:

```
$ nagfor -C=calls sub1.f90 -o sub1.exe
```

- Fortran code:

```
integer, parameter :: x = 12
call sub_test( x )
subroutine sub_test( x )
  integer :: x
  x = 10
end subroutine sub_test
```

- Runtime output:

```
Runtime Error: sub1.f90, line 13: Dummy argument X is
associated with an expression - cannot assign
```

nag®

NAG Compiler Integer Overflow Detection

- Compile command:

```
$ nagfor -C=intovf ovf_test.f90 -o ovf_test.exe
```

- Fortran code:

```
integer :: i, j, k

j = 12312312; k = 12312312
i = 12312312 * j * k
```

- Runtime output:

```
Runtime Error: ovf_test.f90, line 7: INTEGER(int32) overflow
for 12312312 * 12312312
Program terminated by fatal error
```

NAG Compiler Pointer Reference Check (1)

- Compiler command:

```
$ nagfor -C=pointer ptr_test.f90
```

- Fortran code:

```
integer, pointer :: p(:)
integer, target :: i_array(10) = 1
if ( size( i_array ) == 11 ) then
  p => i_array
else
  p => null()
end if
print *, p ! p will be null
```

NAG Compiler Pointer Reference Check (2)

- Runtime output:

```
Runtime Error: ptr_test.f90, line 15: Reference to
disassociated POINTER P
Program terminated by fatal error
```

NAG Compiler Mixed Kind Detection

- Compile command:

```
$ nagfor -c -kind=unique mix_kind.f90
```

- Fortran code:

```
real(kind=REAL64), intent(inout) :: x, y
real(kind=INT32) :: t
t = x; x = y; y = t
```

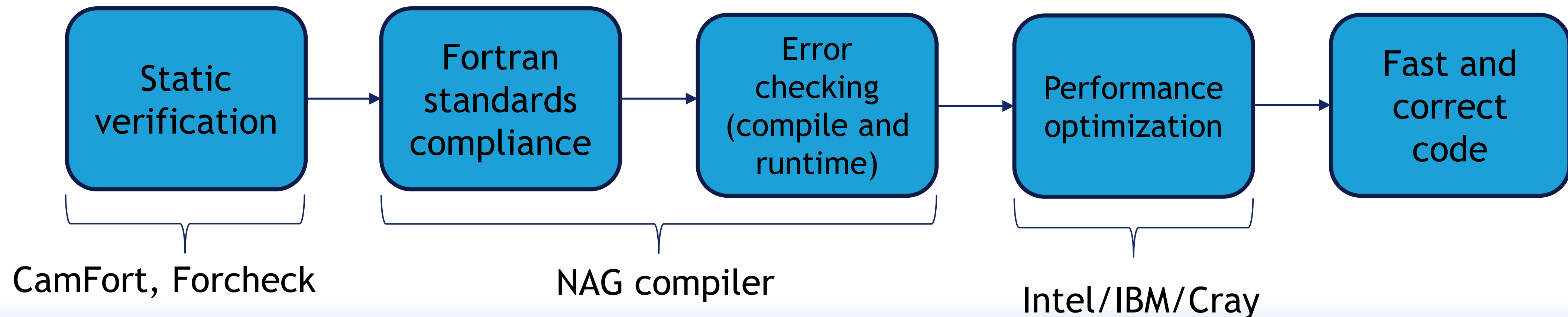
- Compilation error:

```
Error: mix_kind.f90, line 24: KIND value (103) does not
specify a valid representation method
```

nag

Performance Portable Code Workflow

- Performance focused compilers do less error and standards compliance checking;
- Using just one compiler can lock you into that single compiler and could potentially make your code less portable [1];
- The NAG compiler does extensive error and standards checking so you can use it in combination with a more performant compiler.



[1] “Write Portable Code”, B. Hook. No Starch Press, 2005

GNU Makefile

- GNU make is a Linux tool for building Fortran codes in an automated manner. *It only rebuilds codes if any dependencies have changed*;
- It builds a dependency tree to decide what to rebuild, e.g. if source code is newer than the object file/executable, then the target will be rebuilt;
- Code dependencies are specified by the developer;
- It has the ability to build dependencies in parallel resulting in quicker builds. It is used to build the Linux kernel;
- Create a `Makefile` in the same directory as the source code and type the `make` command to build your code. This will build the first target and all dependencies.

Makefile Rules

- Makefiles consist of explicit rules which tell it how to build a target;
- A target can be a code executable, library or module header;

```
target: dependencies
        build commands
```

- *Note that the tab character must precede the build commands*;
- A rule has *dependencies* and the *commands* will build the *target*;
- Compilation and link flags are specified in the Makefile to ensure consistent building of codes;
- Different flags can result in slightly different results in numerical codes, particularly optimization flags.

Compiling a Fortran Module (1)

- When compiling `mesh_mod.F90` which contains a Fortran module called `mesh_mod`, two files are created;
- `mesh_mod.mod` which is a pre-compiled *header module* file which contains Fortran variables and interfaces;
- The path to header module file is specified with `-I` during compilation only, e.g. `-I/usr/library/include`
- `mesh_mod.o` which is an object file which contains all functions and subroutines as *symbols* for linking;

Compiling a Fortran Module (2)

- A number of object files can be bundled into a single library, e.g. `libdep.a`, which is created using the Linux `ar` tool;
- The path to the library is specified using the `-L` flag with `-l` followed by the name of the library, e.g. `-L/home/miahw/dep/lib -ldep`

Automatic Makefile Variables

- The variable `$@` is the target of the rule;
- The variable `$^` contains the names of all prerequisites;
- The variable `$<` contains only the first prerequisite;
- The variable `$?` contain all the prerequisites that are newer than the target;
- To see what commands make will execute without executing them, which is useful for debugging:

```
$ make -n
```

nag

More Makefile Features

- Pattern matching - will compile all files:

```
%.o: %.f90
	nagfor -c -I. $<
```

- However, dependencies between files must be explicitly specified;
- Can include additional files:

```
include variables.mk
```

- Set variables that contain all source files that end in `_mod.f90` and corresponding object files:

```
SOURCES := $(sort $(wildcard *_mod.f90))
OBJECTS := $(SOURCES:.f90=.o)
```

nag

Example Makefile

```
FFLAGS = -O2 -I.                            # add any other compilation flag
LDFLAGS = -L. -L/usr/local/hawk/lib -lhawk # add any other link flag
main.exe: main.o dep1.o dep2.o
      nagfor $^ $(LDFLAGS) -o $@      # (3)

main.o: main.F90 dep1_mod.o dep2_mod.o
      nagfor $(FFLAGS) -I. -c $<      # (2) requires dep1.mod and dep2.mod
dep1_mod.o: dep1_mod.F90
      nagfor $(FFLAGS) -c $<          # (1) also creates dep1.mod
Dep2_mod.o: dep2_mod.F90
      nagfor $(FFLAGS) -c $<          # (1) also creates dep2.mod

.PHONY: clean
clean:
      rm -rf *.o *.mod main.exe
```

Example Makefile (2)

- Typing just `make` will build `main.exe` which is the first and default target;
- Separate targets can be built using `make <target-name>`, e.g. `make dep1.o`;
- Makefile variables are enclosed in brackets, e.g. `${VAR1}`. This can also include Linux environment variables;
- **Can you see a problem with this Makefile? This is related to Fortran modules.**

Makefile and Fortran Modules (1)

- The previous Makefile does not take into account `.mod` files created when modules are compiled;
- If a `.mod` file is deleted, it will not be recreated. Thus, compilation of a Fortran code that uses that module will not compile;
- Two rules are required - output from NAG compiler:

```
$ nagfor =depend -otype=make types_mod.f90
types_mod.mod:  types_mod.f90
types_mod.o:    types_mod.f90
```

Makefile and Fortran Modules (2)

- Solution is to set up make variables:

```
SOURCES := $(sort $(wildcard *_mod.f90))
OBJECTS := $(SOURCES:.f90=.o)
```

- The variable `SOURCES` contains all Fortran module files that end in `_mod.f90`;
- The `OBJECTS` variable contains all object files from module files;

Makefile and Fortran Modules (3)

- Use the NAG compiler dependency tool to create module dependency files which gets created at every make invocation:

```
%.P: %.f90
	nagfor =depend -otype=make $< -o $@.tmp
	grep -vi -E '(netcdf|plplot)' $@.tmp > $@
```

- Second command filters any external library dependencies;
- Create a variable that stores all the module dependencies:

```
DEPS := $(SOURCES:.f90=.P) main_code.P
```

nag

Makefile and Fortran Modules (3)

- Create a single file that contains all the dependencies:

```
Depends: $(DEPS)
	cat $^ > $@
```

- And then finally include this file in the Makefile:

```
include Depends
```

- Then apply the following rule to compile in the correct order:

```
%.o %.mod: %.f90
	nagfor -c $(FFLAGS) $<
```

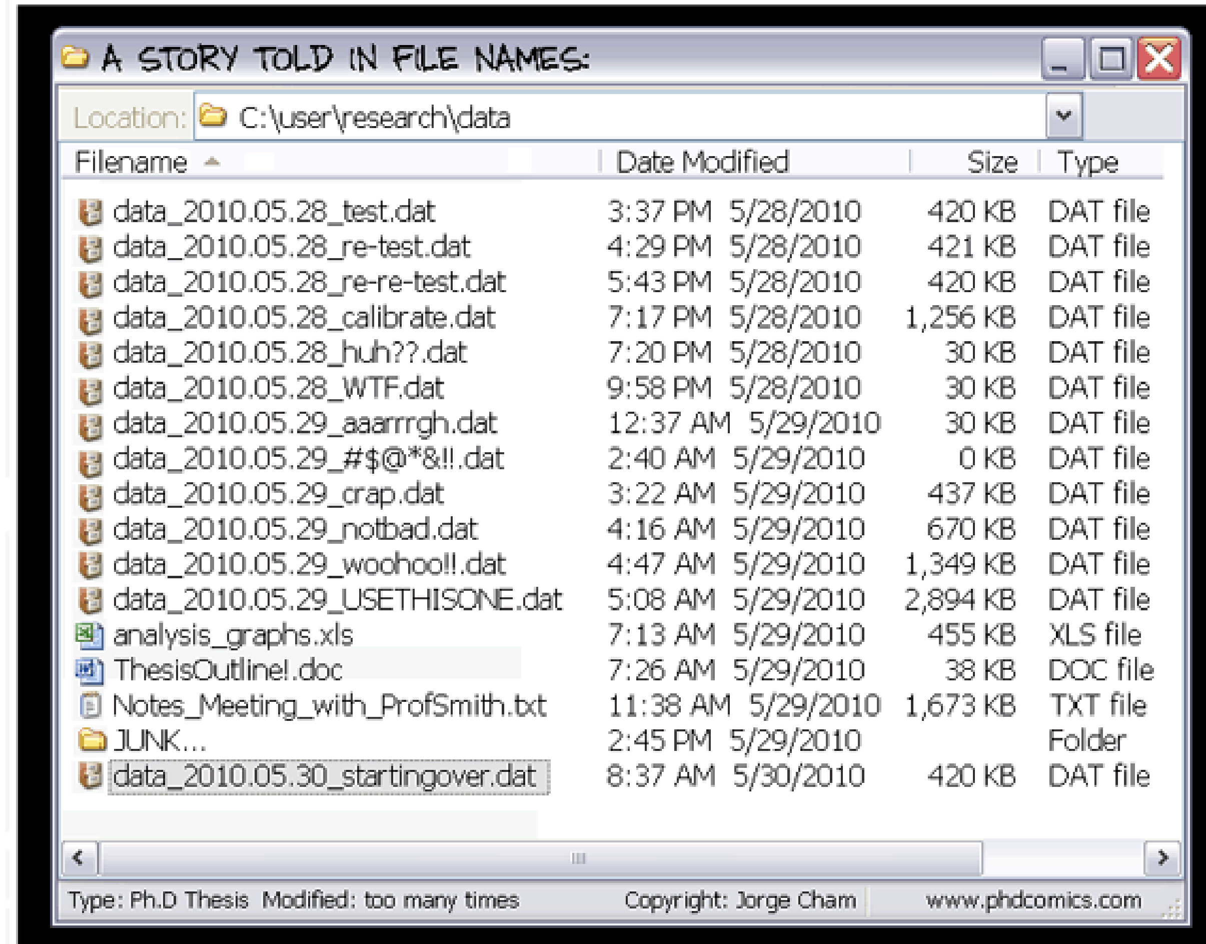
- If the `Depends` file is static, then just created it once.

Parallel Builds Using Makefile

- When writing Makefiles, dependencies must obviously be correctly specified;
- If they are not, you will get link errors resulting in “undefined symbol” messages;
- In addition, parallel builds depend on rule dependencies being correctly defined and only then can you use parallelise builds;
- To parallelise a build with *k* processes, use the command:

```
$ make -j k
```

Data Management

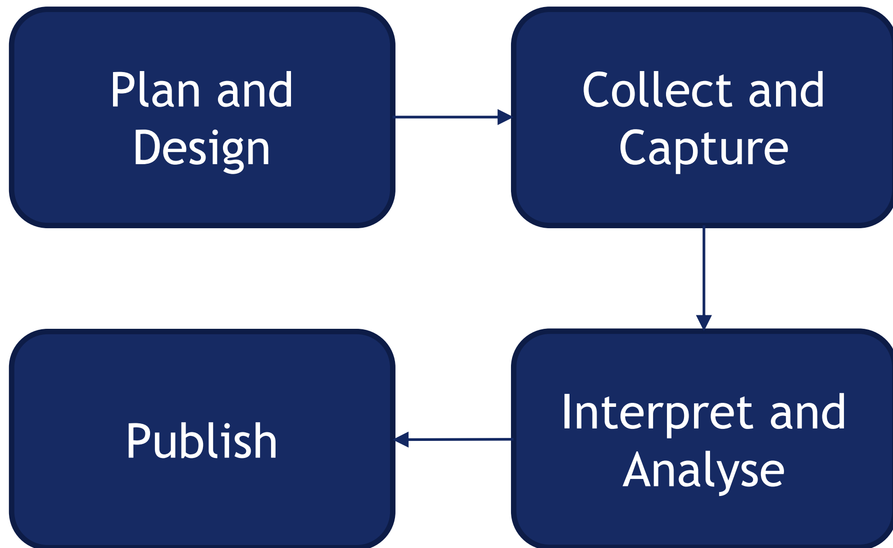


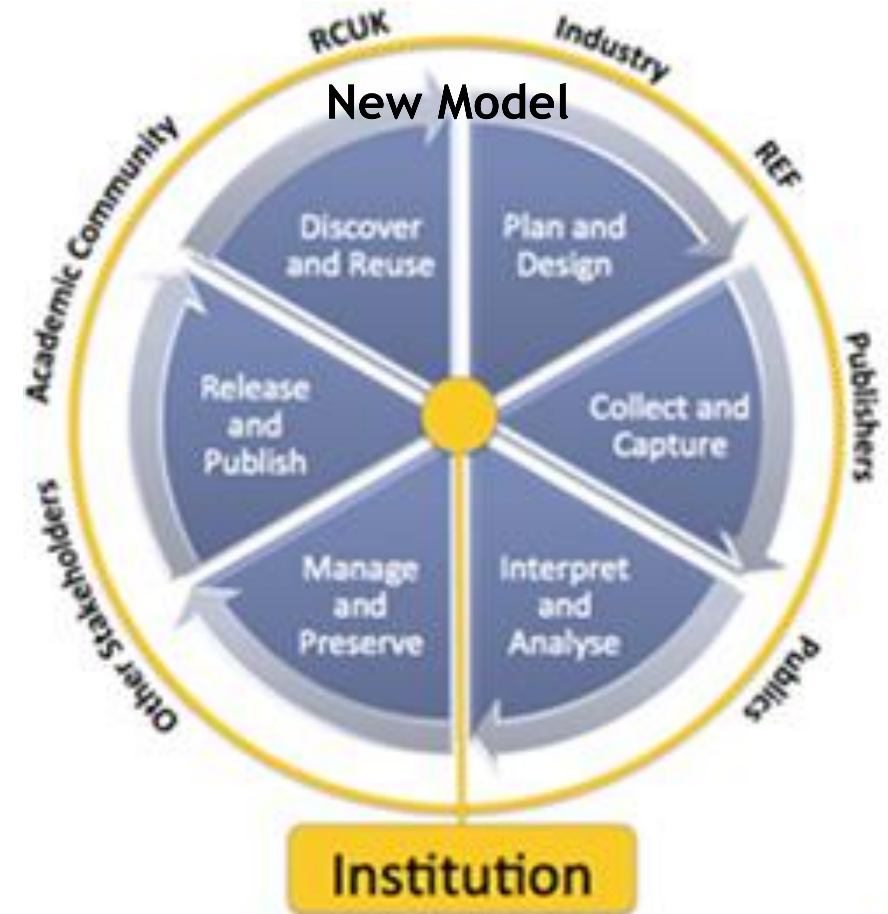
http://phdcomics.com

Data From Simulations

- Computational codes are producing petabytes of data from multiple simulations creating a large number of data sets;
- Data is stored for two reasons: checkpoint/restart for fault resiliency and, visualisation and analysis. *If used for visualisation, consider using single precision as this will halve the size of your data set*;
- Efficient access to single or multiple variables required, e.g. velocity, pressure, temperature;
- The volume of data generated by simulations is proportional to: 1) the FLOPS of the HPC system 2) the memory on the system 3) the underlying computational model used in the code.

Research Data Lifecycle





“Data Management for Researchers”, K. Briney. Pelagic Publishing, 2015

Challenges of Data Management (1)

- Huge number of data sets stored in separate files;
- Sharing datasets with collaborators is difficult due to lack of meta data;
- Large size of data sets and loss of numerical precision due to storing data in incorrect format, e.g. CSV;
- Searching data sets for parameters is difficult also due to lack of meta data;
- Solution: *use a self-describing file format such as NetCDF or HDF5*;
- Python and R bindings are available for NetCDF and HDF5 for data analysis and visualisation;

Challenges of Data Management (2)

- Parallel (MPI) implementations of NetCDF and HDF5 exist;
- Parallel visualisation packages such as VisIt [1] and Paraview [2] are able to read NetCDF and HDF5.

[1] http://visit.llnl.gov
[2] http://www.paraview.org

NetCDF File Format (1)

- Stores data in the form of multi-dimensional arrays;
- Underlying storage is abstracted away from user applications;
- Is portable across many different architectures, hence allows collaboration. It can be read by codes in other programming languages;
- Uses a highly optimized indexing system so data access is direct rather than sequential;
- Applies compression techniques to minimise file sizes;

NetCDF File Format (2)

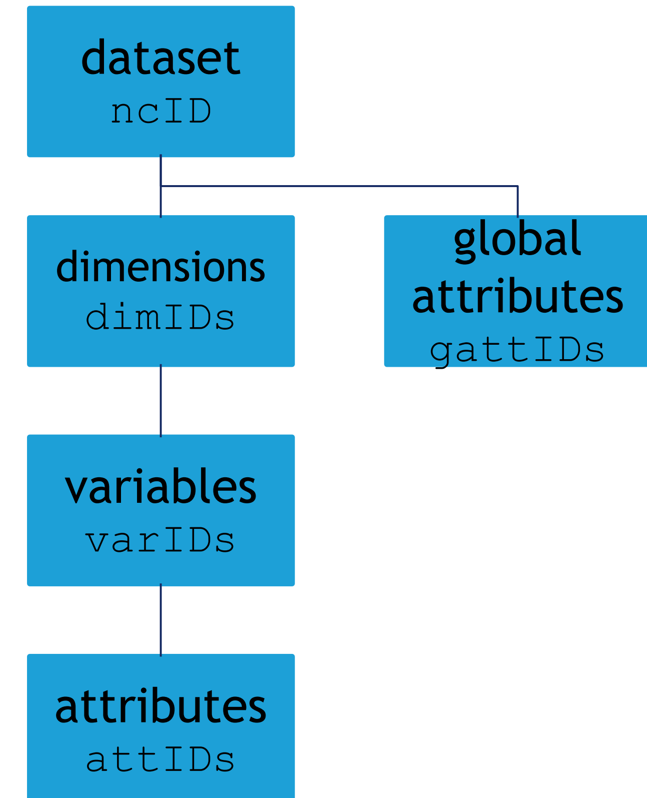
- Uses the IEEE-754 floating point standard for data representation;
- Can store meta-data inside data files so others can understand the data and makes it easier to retrieve at a later date.

Components of NetCDF

- NetCDF *dataset* contains *dimensions*, *variables* and *attributes*. They are all referred to by a unique integer ID value in a Fortran code;
- A dimension has a *name* and *length*, e.g. latitude, *x* dimension. A dimension can have a fixed value or be unlimited, e.g. time varying;
- A variable has a name and is used to store the data, e.g. pressure;
- An attribute is data used to *describe* the variable, e.g. Kelvin, N/m^2;
- Use the attributes to your advantage to describe your experiment and variables. This will help you share your data and avoid repeating the same simulation;
- Every NetCDF function should return `NF90_NOERR` constant.

Common Data Form Language (CDL) Example

```
netcdf dataset1 {
dimensions:
  x = 3, y = 3, time = unlimited;
variables:
  float p(time,x,y);
    p:long_name = "pressure";
    p:units = "N/m^2";
data:
  p = 0.1, 0.2, 0.3,
      1.2, 3.4, 3.2,
      3.2, 2.0, 1.9;
}
```



Creating a NetCDF Dataset

```
NF90_CREATE        ! create dataset. enter define mode
   NF90_DEF_DIM    ! define dimensions
   NF90_DEF_VAR    ! define variables
   NF90_PUT_ATT    ! define attributes

NF90_ENDDEF        ! end define mode. enter data mode
   NF90_PUT_VAR    ! write your data
NF90_CLOSE         ! close your data set
```

Reading a NetCDF Dataset

```
NF90_OPEN              ! open data set. enter data mode
   NF90_INQ_DIMID      ! enquire to obtain dimension IDs
   NF90_INQ_VARID      ! enquire to obtain variable IDs

   NF90_GET_ATT        ! get variable attributes
   NF90_GET_VAR        ! get variable data
NF90_CLOSE             ! close data set
```

Creating a NetCDF Dataset

```
function NF90_CREATE( path, cmode, ncid )
```

- `path` to dataset including filename, e.g. `/home/miahw/data.nc`;
- `cmode` is either `NF90_CLOBBER` or `NF90_NOCLOBBER`. Former will overwrite any existing file and latter will return an error;
- `ncid` is a unique ID for dataset. Any dataset related operations should use this integer.
- To close a data set, simply invoke:

```
function NF90_CLOSE( ncid )
```

Opening a NetCDF Dataset

```
function NF90_OPEN( path, omode, ncid )
```

- `path` to dataset including filename, e.g. `/home/miahw/data.nc`;
- `omode` is `NF90_NOWRITE` by default or `NF90_WRITE`. Former will read an existing file and latter allows *appending* to a file;
- `ncid` is a unique ID for dataset. Any dataset related operations should use this integer.

Creating a NetCDF Dimension

- Dimensions are created when in defined mode and have a name and a unique identifier;
- They can be constant, e.g. number of cells in *x*-direction;
- Or they can be `NF90_UNLIMITED`, e.g. time steps.

```
function NF90_DEF_DIM( ncid, name, len, dimid )
```

- `ncid` - ID of dataset;
- `name` - name of dimension;
- `len` - length of dimension;
- `dimid` - the returned ID of the identifier which is assigned by the function.

Creating a NetCDF Variable (1)

- Variables are created when in defined mode and have a name and a unique identifier;
- They can be a scalar or a multi-dimensional array. The dimension IDs are used to define the number and length of dimensions.

```
function NF90_DEF_VAR( ncid, name, xtype, dimids, varid )
```

- `ncid` - ID of dataset;
- `name` - name of variable;
- `xtype` - type of variable;
- `dimids` - the IDs of created dimensions, e.g. `[ dimid1, dimid2 ]`
- `varid` - the returned ID of the variable;

nag®

Creating a NetCDF Variable (2)

- The data type `xtype` may be one of the listed mnemonics:

Fortran Mnemonic	Bits
`NF90_BYTE`	8
`NF90_CHAR`	8
`NF90_SHORT`	16
`NF90_INT`	32
`NF90_FLOAT` or `NF90_REAL4`	32
`NF90_DOUBLE` or `NF90_REAL8`	64

Creating a NetCDF Attribute (1)

- An attribute is data about data, i.e. metadata, and is used to describe the data;
- It has a name and a value.

```
function NF90_PUT_ATT( ncid, varid, name, value )
```

- `ncid` - ID of dataset;
- `varid` - ID of variable;
- `name` - name of attribute which is a string;
- `value` - value of attribute which is a string;

Creating a NetCDF Attribute (2)

- Typical attributes stored for variables: `units`, `long_name`, `valid_min`, `valid_max`, `FORTRAN_format`;
- Use any attribute that is useful for describing the variable;
- Global attributes for dataset can also be stored by providing `varid = NF90_GLOBAL`;
- Typical global attributes: `title`, `source_of_data`, `history` (array of strings), `env_modules`, `doi`;
- *Use any attribute that is useful for describing the dataset as this will increase data sharing and collaboration!*
- Further metadata can be included in the file name.

Writing and Reading NetCDF Data

- Once the IDs have been set up, the data can then be written;

```
function NF90_PUT_VAR( ncid, varid, values, start, count )
```

- `ncid` - ID of dataset;
- `varid` - variable ID
- `values` - the values to write and can be any rank;
- `start` - array of start values and `size( start ) = rank( values )`
- `count` - array of count values and `size( count ) = rank( values )`
- Last two arguments are optional;
- The read function `NF90_GET_VAR` has the same argument set.

NetCDF Write Example

```
integer, dimension(NX,NY) :: data
integer :: ierr, ncid, x_dimid, y_dimid, varid
ierr = NF90_CREATE( "example.nc", NF90_CLOBBER, ncid )
data(:, :) = 1 ! entering define mode

ierr = NF90_DEF_DIM( ncid, "x", NX, x_dimid )
ierr = NF90_DEF_DIM( ncid, "y", NY, y_dimid )
ierr = NF90_DEF_VAR( ncid, "data", NF90_INT, [ x_dimid, y_dimid ], &
     &                    varid )
ierr = NF90_ENDDEF( ncid ) ! end define mode and enter data mode

ierr = NF90_PUT_VAR( ncid, varid, data ) ! write data
ierr = NF90_CLOSE( ncid )
```

NCO - NetCDF Commands (1)

- `ncdump` - reads a binary NetCDF file and prints the CDL (textual representation) to standard out;
- `ncgen` - reads the CDL and generates a binary NetCDF file;
- `ncdiff` - Calculates the difference between NetCDF files;
- `ncks` - ability to read subsets of data much like in SQL. Very powerful tool for data extraction;
- `ncap2` - arithmetic processing of NetCDF files;
- `ncatted` - NetCDF attribute editor. Can append, create, delete, modify and overwrite attributes.

NCO - NetCDF Commands (2)

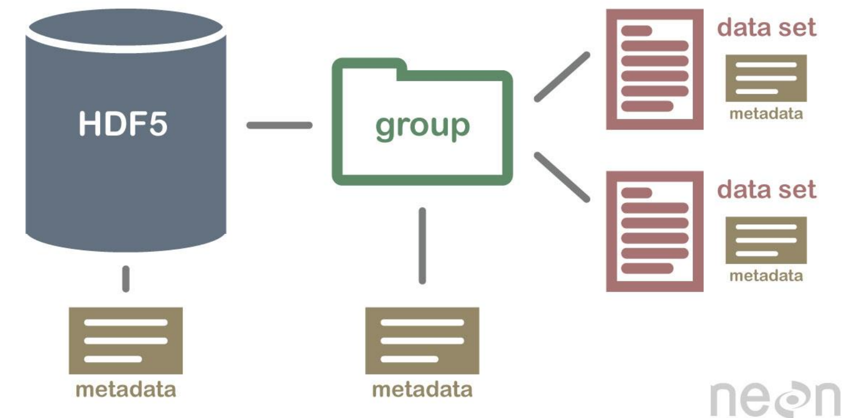
- `ncrename` - renames dimensions, variables and attributes in a NetCDF file;
- `ncra` - averages record variables in arbitrary number of input files;
- `ncwa` - averages variables in a single file over an arbitrary set of dimensions with options to specify scaling factors, masks and normalisations;
- `nccopy` - converts a NetCDF file, e.g. version 3 to version 4. It can also compress data or changing the chunk size of the data.

HDF5 File Format

- HDF5 is a data model and file format, and provides an API to use within application codes;
- It is similar to NetCDF in that it allows binary data to be stored and is fully portable to other architectures and programming languages;
- Datasets can be arranged in a hierarchical manner;
- Self-describing data format and allows metadata to be stored;
- Efficiently stores data and allows direct access to data;
- Has been developed for over 25 years and widely used by the scientific community;
- More complicated that NetCDF.

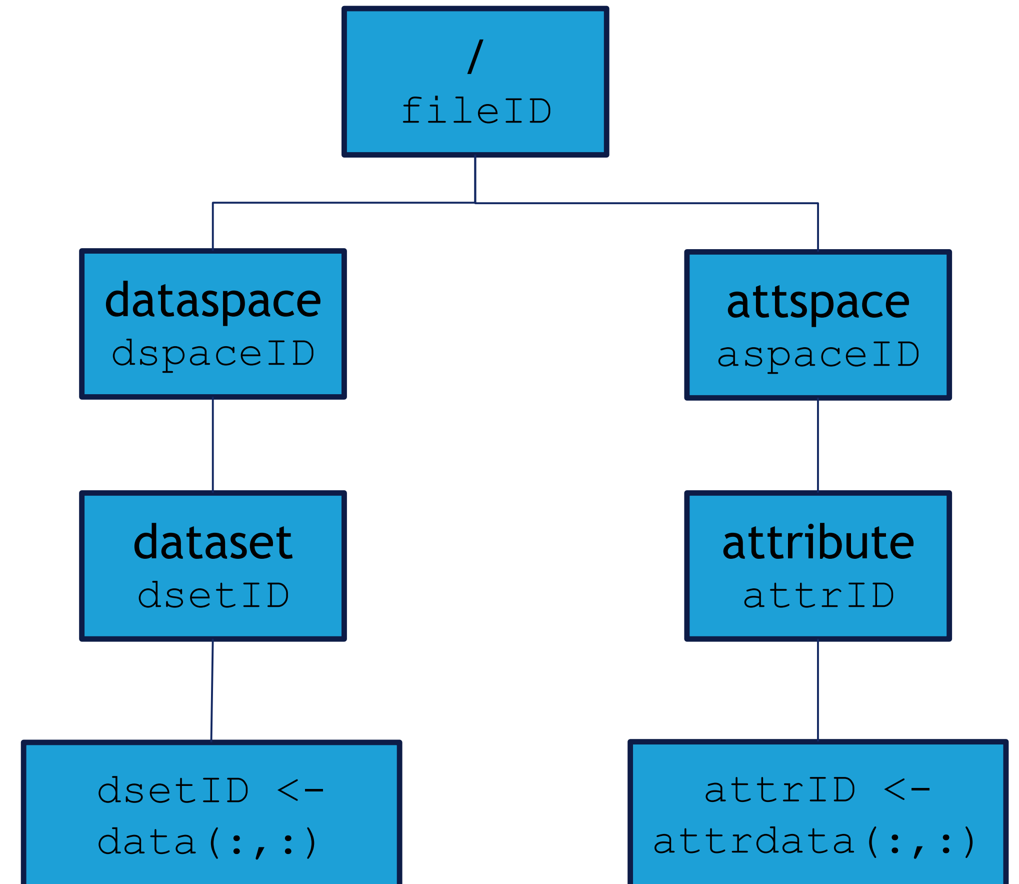
HDF5 Data Model

- **File**: contains all groups and datasets, and at least one group - root /
- **Dataset**: multi-dimensional data array;
- **Group**: a set of links to datasets or other groups;
- **Link**: reference to a dataset or group;
- **Attribute**: metadata for dataset or group;



HDF5 Dataset Definition Language

```
<dataset> ::=
   DATASET "<dataset_name>" {
       <datatype>
       <dataspace>
       <data>
       <dataset_attribute>*
}
<datatype> ::= DATATYPE { <atomic_type> }
<dataspace> ::= DATASPACE {
     SIMPLE <current_dims> / <max_dims> }
<dataset_attribute> ::= <attribute>
```



Creating a HDF5 Dataset

```
H5OPEN_F                    ! initialise HDF5
  H5FCREATE_F               ! create file
  H5SCREATE_SIMPLE_F        ! create dataspace
  H5DCREATE_F               ! create dataset
  H5DWRITE_F                ! write data

  H5DCLOSE_F                ! close dataset
  H5SCLOSE_F                ! close dataspace
  H5FCLOSE_F                ! close file
H5CLOSE_F                   ! finalise HDF5
```

Reading a HDF5 Dataset

```
H5OPEN_F          ! initialise HDF5
  H5FOPEN_F       ! open file
  H5DOPEN_F       ! open dataset
  H5DREAD_F       ! read dataset

  H5DCLOSE_F      ! close dataset
  H5FCLOSE_F      ! close file
H5CLOSE_F         ! finalise HDF5
```

HDF5 Write Example

```
integer(kind = HID_T) :: file_id, dset_id, dspace_id, rank = 2
integer(kind = HSIZE_T), dimension(1:2) :: dims = [ 4, 6 ]

call H5OPEN_F( ierr )
call H5FCREATE_F( "dsetf.h5", H5F_ACC_TRUNC_F, file_id, ierr )
call H5SCREATE_SIMPLE_F( rank, dims, dspace_id, ierr )
call H5DCREATE_F( file_id, "dset", H5T_NATIVE_INTEGER, dspace_id, &
     &            dset_id, ierr )
call H5DWRITE_F( dset_id, H5T_NATIVE_INTEGER, dset_data, dims, ierr )

call H5DCLOSE_F( dset_id, ierr ); call H5SCLOSE_F( dspace_id, ierr )
call H5FCLOSE_F( file_id, ierr )
call H5CLOSE_F( ierr )
```

In-Memory Visualisation with PLplot (1)

- In-memory visualisation can visualise the data whilst it is in memory and does not require the data to be stored on disk;
- This subsequently saves disk space and time as data reading/writing is prevented, thus avoiding the I/O bottleneck;
- PLplot [1] is a scientific graphics library with bindings for Fortran 90;
- It can create standard x-y plots, semi-log plots, log-log plots, contour plots, 3D surface plots, mesh plots, bar charts and pie charts;
- Formats supported are: GIF, JPEG, LaTeX, PDF, PNG, PostScript, SVG and Xfig;

[1] http://plplot.sourceforge.net/

In-Memory Visualisation with PLplot (2)

- Visualisation is done within the Fortran code and does not require an additional script. Quicker to produce quality graphs which can be used for publication;
- It is also used to test your models and configurations whilst the simulation is executing;
- If your solution does not converge or produces unphysical effects then the simulation job can be terminated, thus saving days or weeks of simulation time;
- It is not meant to compete with any of the other major visualisation packages such as GNUPlot or Matplotlib.

PLplot Subroutines (1)

- Load the Plplot Fortran module:

```
use plplot
```

- The output format needs to be specified [2]:

```
call PLSDEV( 'pngcairo' )
```

- The image file name needs to be specified:

```
call PLSFNAM( 'output.png' )
```

- The library needs to be initialised:

```
call PLINIT( )
```

- Specify the ranges, axes control and drawing of the box:

```
call PLENV( xmin, xmax, ymin, ymax, justify, axis )
```

[2] Other formats supported are: `pdfcairo pscairo epscairo svgcairo`

PLplot Subroutines (2)

- Specify the x- and y-labels and title:

```
call PLLAB( ‘x’, ‘y’, ‘plot title’)
```

- Draw line plot from one-dimensional arrays:

```
call PLLINE( x, y )
```

- Finalise PLplot:

```
call PLEND( )
```

- To compile and link:

```
$ nagfor -c -I/plplot/modules graph.F90
$ nagfor graph.o -L/plplot/lib -lplplotfortran -lplplot \
            -o graph.exe
```

FFMPEG

- FFMPEG is a utility to convert between audio and video formats;
- In this workshop, it will be used to create a movie file from a list of images which were created by PLplot;
- To create an MP4 movie from a list of images, e.g. `image_01.png`, `image_02.png`, use:

```
$ ffmpeg -framerate 1/1 -f image2 -i image_%*.png video.mp4
```

- FFMPEG has many options and has a collection of codecs;
- Movies can then be embedded into presentations.

End of Day 1 - Start Exercises

- Exercise code is at `fmw_exercises/src/fd1d_heat_explicit.f90`
- Presentation can be found at - also copy over to your laptop/desktop
  `fmw_exercises/FortranModernisationWorkshop.pdf`
- To copy in Linux, type (in one line):

```
$ scp
username@training.nag.co.uk:~/fmw_exercises/exercises.pdf .
```

- Replace *username* with the provided user name.

Day Two Agenda

- Introduction to parallelisation in MPI, OpenMP, Global Arrays and CoArrays. GPU programming using CUDA Fortran and OpenACC;
- Parallel I/O using HDF5 and NetCDF;
- Introduction to the NAG numerical library;
- Fortran interoperability with R, Python and C.

Fortran Syntax Checkers for Linux Editors

- Fortran syntax checkers also exists for traditional Linux editors such as vim and Emacs which checks syntax as you type;
- Idea is to identify syntax violations as quickly as possible instead of waiting for a build failure;
- Syntax checkers increase the productivity of users by providing a quick feedback on Fortran language violations;
- For Emacs users, the Flycheck syntax checker is available at [1];
- For vim users, the Syntastic plugin is available [2].

[1] http://www.flycheck.org/
[2] https://github.com/scrooloose/syntastic

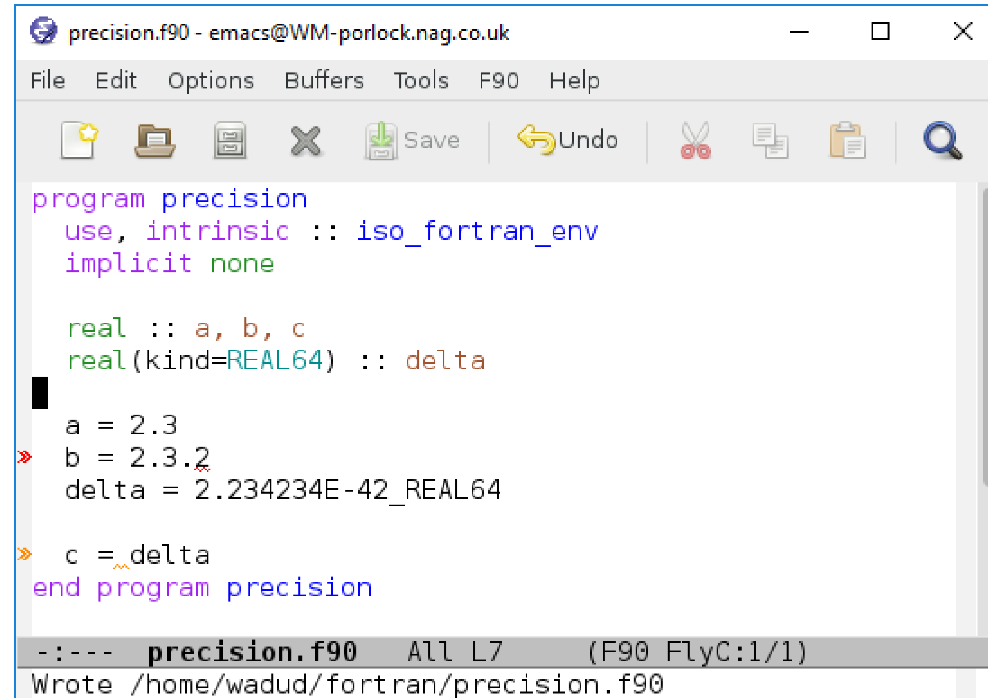
Flycheck for Fortran (1)

- In your `~/.emacs` file, include the following configuration:

```
(setq flycheck-gfortran-language-standard "f2008")
(setq flycheck-gfortran-warnings '("all" "unused"))
(setq flycheck-gfortran-args '("-Wunderflow" "-Wextra"))
(setq flycheck-gfortran-include-path '("../include"))
```

- Flycheck uses the installed GNU Fortran compiler for syntax checking with the above flags.

Flycheck for Fortran (2)



- Dark red arrows and underline show compilation *errors*;
- Orange arrows and underline shows compiler *warnings*.

Fortran 90 Emacs Settings

- The following settings are required in the `~/.emacs` file:

```
(setq f90-do-indent 2)
(setq f90-if-indent 2)
(setq f90-type-indent 2)
(setq f90-program-indent 2)
(setq f90-continuation-indent 4)
(setq f90-comment-region "!!$")
(setq f90-indented-comment-re "!")
```

Emacs Fortran Navigation

`CTRL-c CTRL-n` Move to the beginning of the next statement;

`CTRL-c CTRL-p` Move to the beginning of the previous statement;

`CTRL-c CTRL-e` Move point forward to the start of the next code block;

`CTRL-c CTRL-a` Move point backward to the previous block;

`CTRL-ALT-n` Move to the end of the current block

`CTRL-ALT-p` Move to the start of the current code block

Syntastic for VIM (1)

- In your `~/.vimrc` file, add the following settings:

```
let g:syntastic_fortran_compiler = 'gfortran’
```
(or `ifort`)

```
let g:syntastic_fortran_compiler_options = ‘-Wall -Wextra’
let g:syntastic_fortran_include_dirs = [ ‘/moddir1’,
‘/moddir2’]
```

- Syntastic only checks the syntax once you have saved the file.

Syntastic for VIM (2)

```
wadudm@wm-porlock:~
File  Edit  View  Search  Terminal  Help
    real :: a

>>  a = 1.2.3
  end program hello_world
  ~
  ~
  ~
  ~
  ~
  ~
  ~
[Syntax: line:5 (1)]
fortran/hello_world.f90|5 col 11 error|  Invalid character in name at (1)
~
~
~
~
~
~
~
~
~
[Location List] :SyntasticCheck gfortran (fortran)          1,1          All
"fortran/hello_world.f90" 6L, 85C written
```

Asynchronous Lint Engine (ALE)

- ALE [1] is another checker for VIM which checks syntax on the fly;
- Add the following lines in your `~/.vimrc` file:

```
let g:ale_fortran_gcc_use_free_form = 1
let g:ale_fortran_gcc_executable = ‘gfortran’
let g:ale_fortran_gcc_options = ‘-Wall -Wextra’
```

[1] https://github.com/w0rp/ale

Asynchronous Lint Engine Screenshot

Terminal - miahw@bedale:~/fortran

File Edit View Terminal Tabs Help

```
program for_test
  implicit none
  real :: a

  print *, 'hello'

>>  a = 1.2.3
end program for_test
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
Invalid character in name at (1)                        7,11         All
```

Fortran JSON

- Fortran JSON [1] offer a convenient way to read configuration files for scientific simulations;
- Do not use JSON for storing data - use either NetCDF or HDF5. Its purpose here is only for simulation configuration parameters;
- JSON format was popularised by JavaScript and is used by many programming languages;
- It is a popular format to exchange data and is beginning to replace XML and is human readable;
- It is strongly recommended to store simulation configuration parameters as the simulation can be reproduced.

[1] https://github.com/jacobwilliams/json-fortran

Example JSON file (`config.json`)

```
{
“config1":
{"major": 2,
 "string": "2.2.1",
 "tol": 3.2E-8,
 "max": 34.23}
}
```

Reading JSON File in Fortran (1)

```
use json_module
use, intrinsic :: iso_fortran_env
implicit none
type(json_file) :: json
logical :: found
integer :: i
real(kind=REAL64) :: tol, max
character(kind=json_CK, len=:), allocatable :: str
```

Reading JSON File in Fortran (2)

```
call JSON%INITIALIZE( )
call JSON%LOAD_FILE(filename = 'config.json' )
call JSON%GET( 'config1.major', i, found )
call JSON%GET( 'config1.string', str, found )
call JSON%GET( 'config1.tol', tol, found )
call JSON%GET( 'config1.max', max, found )
call JSON%DESTROY( )
```

Fortran Command Line Arguments Parser (1)

- The Fortran command line arguments parser (FLAP) [1] allows command line arguments to be processed;
- It is similar to the Python *argparse* command line parser and is more elegant than the `get_command_argument( )` intrinsic subroutine;

```
use flap
implicit none
type(command_line_interface) :: cli
integer                      :: ierr, i
real                         :: tol
```

[1] https://github.com/szaghi/FLAP

nag®

Fortran Command Line Arguments Parser (2)

```
call CLI%INIT(description = 'minimal FLAP example')
call CLI%ADD( switch = '--int', switch_ab = '-i', &
              help = 'an integer (number of intervals)', &
              required = .true., act = 'store', error = ierr )
call CLI%ADD( switch = '--tol', switch_ab = '-t', &
              help = 'a real (tolerance)', required = .true., &
              act = 'store', error = ierr )
call CLI%GET( switch = '-i', val = i, error = ierr )
call CLI%GET( switch = '-t', val = tol, error = ierr )
```

Parallel Programming in Fortran (1)

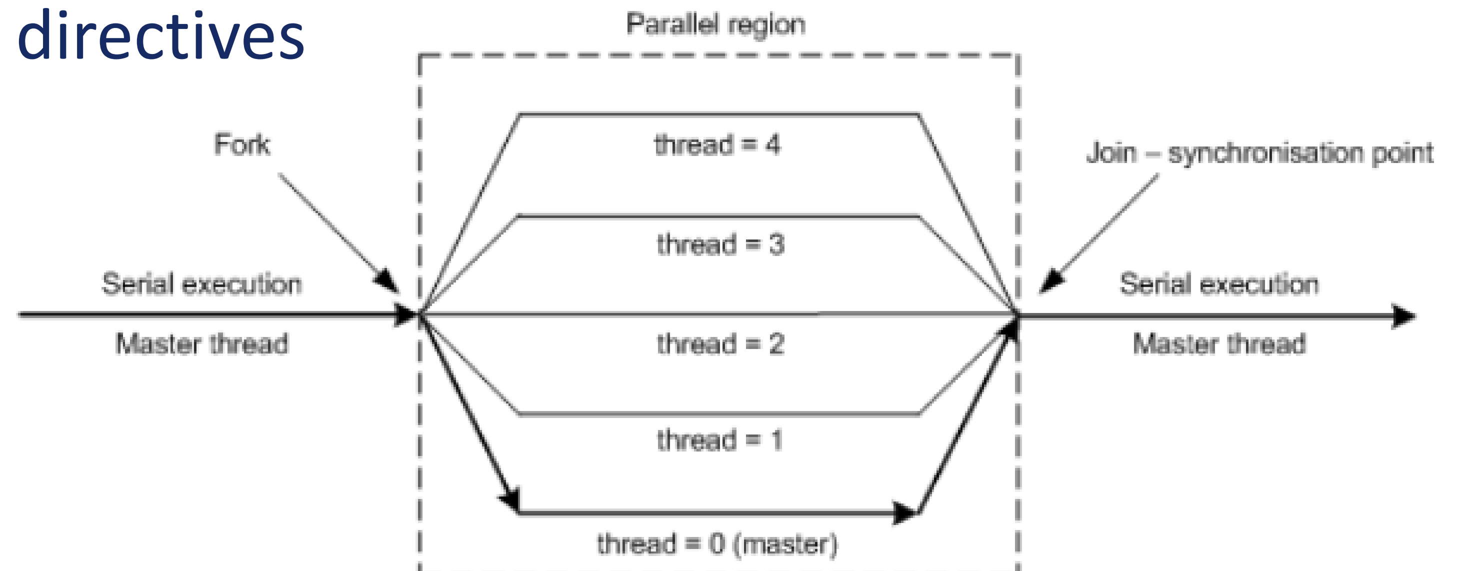
- FORTRAN 77 was a simple standard which compilers could exploit to create optimised code;
- Modern Fortran is likely to cause some slowdown as it provides newer features;
- To offset this slowdown in your code, you can parallelise;
- Computational codes usually take a long time to complete, hence the need for parallelism;
- In addition, problem sizes are increasing, hence the need for more memory;

Parallel Programming in Fortran (2)

- There are a number of ways to parallelise:
  - Shared memory (OpenMP)
  - Distributed memory (CoArray Fortran, GA, MPI)
  - GPU (OpenACC, CUDA Fortran)
  - Vectorization

Shared Memory (1)

- OpenMP (Open Multi-Processing)
- Parallel *across cores within node* (better to limit to NUMA node)
- Spawns threads and joins them again
- Surround code blocks with directives



www.openmp.org

Shared Memory (2)

```
01 use omp_lib
02 !$omp parallel default(shared), private(threadN)
03 !$omp single
04   nThreads = omp_get_num_threads()
05 !$omp end single
06   threadN = omp_get_thread_num()
07   print *, “I am thread”, threadN, “of”, nThreads
08 !$omp end parallel
```

Shared Memory (3)

```
01 !$omp parallel default(none), shared(n, a, X, Y, Z), private(i)
02 !$omp do
03   do i = 1, n
04     Z(i) = a * X(i) + Y(i)
05   end do
06 !$omp end do
07 !$omp end parallel
```

Shared Memory (4)

```
01 !$omp parallel default(none), shared(a, X, Y, Z)
02 !$omp workshare
03   Z(:) = a * X(:) + Y(:)
04 !$omp end workshare
05 !$omp end parallel
```

OpenMP Do Construct

- OpenMP parallel schedule and data decomposition can also be controlled by the `schedule` clause:

```
!$omp do schedule( type[, chunk_size ] )
```

where *`type`* is `static` (default), `dynamic`, `guided`, `auto` or `runtime`, and `chunk_size` is the number of iterations each thread will execute;

- The OpenMP do construct has much better support than the workshare construct, thus may be beneficial to use Fortran do loops rather than array operations.

Distributed memory

- Single Program Multiple Data (SPMD);
- Parallel *across nodes* and/or cores within node;
- Each process has its unique memory space and there is no sharing of memory between processes;
- Data must be explicitly communicated with other processes via send and receive calls;

- MPI (Message Passing Interface) [1]

- PGAS (Partitioned Global Address Space) [2]:
  - CoArray Fortran and GA (Global Arrays) - similar to Unified Parallel C (UPC)

[1] www.mpi-forum.org
[2] www.pgas.org

Detour: Example Code

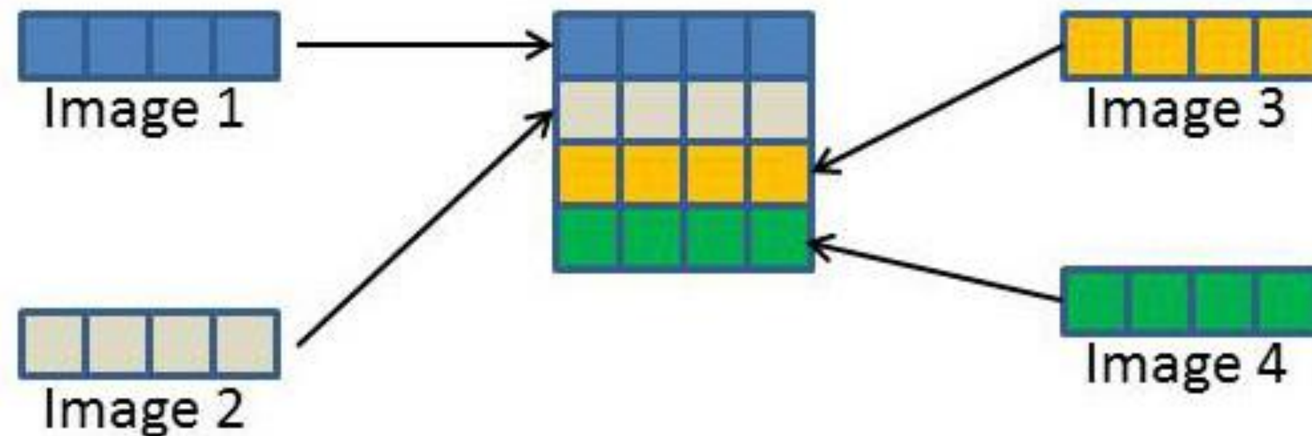
```
01 subroutine axpy(n, a, X, Y, Z)
02 implicit none
03 integer :: n
04 real :: a
05 real :: X(*), Y(*), Z(*)
06 integer :: I
07   do i = 1, n
08     Z(i) = a * X(i) + Y(i)
09   end do
10 end subroutine axpy
```

CoArrays

- *Shared* and *distributed* memory modes (compile time dependent);
- Each process is called an *image* and communication between images is *single sided* and *asynchronous;*
- An image accesses remote data using CoArrays;
- Fortran is the only language that provides distributed memory parallelism as part of the standard (Fortran 2008);
- *Supposed* to be interoperable with MPI;
- Coarrays have **corank**, **cobounds**, **coextent** and **coshape**. Indices used in coarrays are known as cosubscripts which maps to an image index.

CoArray Declaration (1)

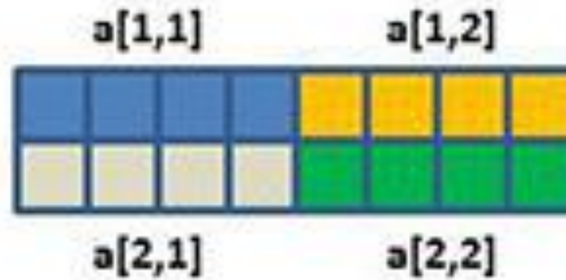
```
01 real, dimension(4), codimension[*] :: mat
$ aprun -n 4 ./caf_matrix.exe
```



- Coshape of coarray is `mat(:)[1:m]` where m is the number of images **which is specified at runtime**. In this example, it is 4;

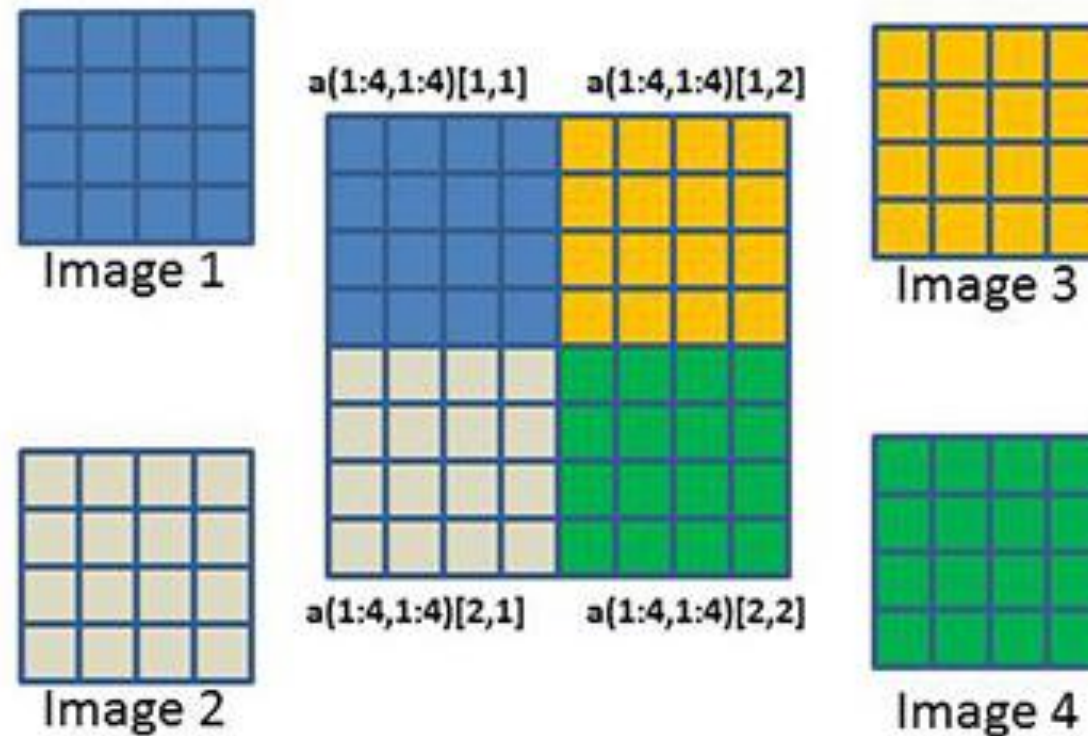
CoArray Declaration (2)

```
01 real, dimension(4), codimension[2, *] :: mat
$ aprun -n 4 ./caf_matrix.exe
```



CoArray Declaration (3)

```
01 real, dimension(4, 4), codimension[2, *] :: bmat
$ aprun -n 4 ./caf_bmatrix.exe
```



CoArray AXPY Example

```
01 real :: a_I[*]
02 real, allocatable :: X_I(:)[:], Y_I(:)[:], Z_I(:)[:]
03 integer :: n_I
04   n_I = n / num_images()
05   allocate( X_I(n_I) ); allocate( Y_I(n_I) ); allocate( Z_I(n_I) )
06   if ( this_image() == 1 ) then
07     do i = 1, num_images()
08       a_I[i] = a
09       X_I(:)[i] = X((i-1)*n_I+1:i*n_I)
10       Y_I(:)[i] = Y((i-1)*n_I+1:i*n_I)
11     end do
12   end if
13   sync all
14   call axpy(n_I, a_I, X_I, Y_I, Z_I)
15   if ( this_image() == 1 ) then
16     do i = 1, num_images()
17       Z((i-1)*n_I+1:i*n_I) = Z_I(:)[i]
18     end do
19   end if
```

Fortran 2018 Collectives (1)

- New collective subroutines:

```
co_max( A [, result_image, stat, errmsg ] )
co_min( A [, result_image, stat, errmsg ] )
co_sum( A [, result_image, stat, errmsg ] )
```

- The above are collective calls and `A` must be the same shape and type;
- If `result_image` is supplied, it is returned to the specified image. It is undefined on all other images;
- `stat` and `errmsg` are returned and contain the status of the call;

Fortran 2018 Collectives (2)

- Broadcasts `a` from image `source_image` to all other images:

```
co_broadcast( a, source_image[, stat, errmsg ] )
```

- Reduction operation where `operation` is a pure function with exactly two arguments and the result is the same type as `A`:

```
co_reduce( a, operation[, result_image, stat, errmsg ] )
```

- If an image has failed, `stat=ierr` will be `STAT_FAILED_IMAGE`

CoArray Teams (1)

- Create new teams:

```
form team ( team_num, team_variable )
```

- `team_num` is an integer and `team_variable` is of `team_type`;
- To change to another team:

```
change team ( new_team )
  ! statements executed with the new_team
end team
```

- Get the team number use `team_number( [ team ] )`

CoArray Teams (2)

- Below is an example taken from the 2018 standards document:

```
change team (team_surface_type)
  select case (team_number( ))
  case (LAND) ! compute fluxes over land surface
    call compute_fluxes_land(flux_mom, flux_sens, flux_lat)
  case (SEA) ! compute fluxes over sea surface
    call compute_fluxes_sea(flux_mom, flux_sens, flux_lat)
  case (ICE) ! compute fluxes over ice surface
    call compute_fluxes_ice(flux_mom, flux_sens, flux_lat)
  end select
end team
```

CoArray Teams (3)

- More intrinsic functions:

`this_image( team )` - returns the image index from `team`

`this_image( corray[, team] )` - returns a rank-one integer array holding the sequence of cosubscript values for coarray

`this_image( coarray, dim[, team] )` - returns the value of cosubscript `dim` in the sequence of cosubscript values for `coarray` that would specify an executing image, i.e. `this_image(coarray)[dim]`

`num_images( team )` - returns the number of images of `team`

`num_images( team_number )` - returns the number of images of `team_number`

nag

Fortran 2018 Fault Tolerance

- Returns a list of images (integers of `KIND` type) that have failed or stopped:

```
failed_images( [ team, kind ] )
stopped_images( [ team, kind ] )
```

- The developer has to manually deal with image failures, e.g. read from the previous checkpoint and restart calculations;
- The argument `team` is of `team_type`;
- Returns `STAT_FAILED_IMAGE` or `STAT_STOPPED_IMAGE`:

```
image_status( image[, team] )
```

CoArrays Locks and Critical (1)

- Supports critical sections which can also be labelled:

```
UPDATE: critical
  i[1] = i[1] + 1
end critical UPDATE
```

- Supports locking to protect shared variables:

```
use iso_fortran_env
type(lock_type) :: lock_var[*]
lock( lock_var[1] )
i[1] = i[1] + 1
unlock( lock_var[1] )
```

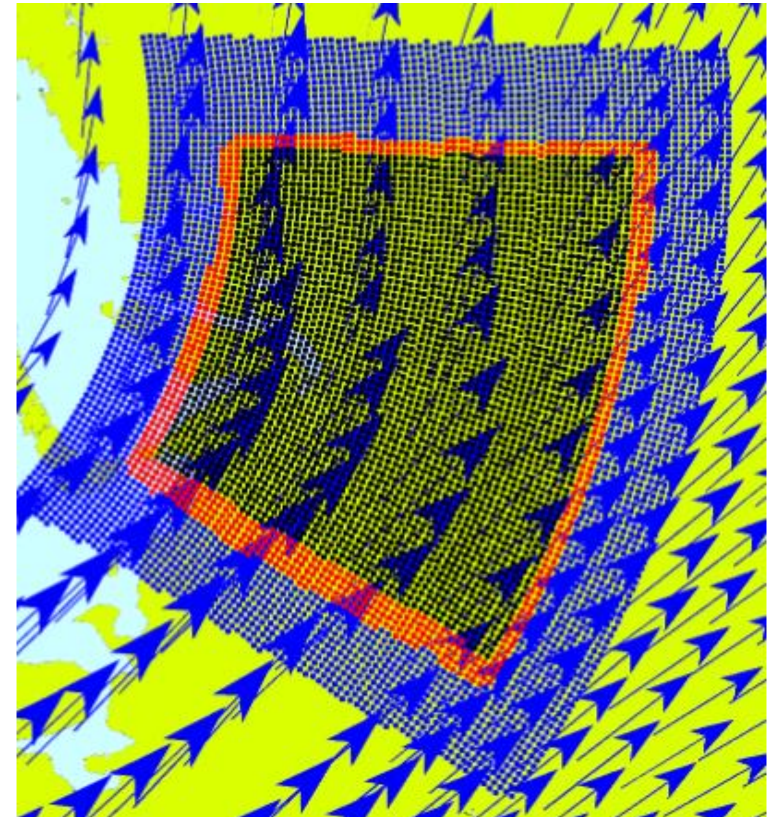
CoArrays Locks and Critical (2)

- Can check to see if lock was acquired:

```
logical :: gotit

lock( lock_var[1], acquired_lock = gotit )
if ( gotit ) then
  ! I have the lock
else
  ! I do not have the lock - another image does
end if
```

CoArray IFS Example

```
!$OMP PARALLEL DO SCHEDULE(DYNAMIC,1) PRIVATE(JM,IM,JW,IPE,ILEN,ILENS,IOFFS,IOFFR)
DO JM=1,D%NUMP
  IM = D%MYMS(JM)
  CALL LTINV(IM,JM,KF_OUT_LT,KF_UV,KF_SCALARS,KF_SCDERS,ILEI2,IDIM1,&
    & PSPVOR,PSPDIV,PSPSCALAR ,&
    & PSPSC3A,PSPSC3B,PSPSC2 , &
    & KFLDPTRUV,KFLDPTRSC,FSPGL_PROC)
  DO JW=1,NPRTRW
    CALL SET2PE(IPE,0,0,JW,MYSETV)
    ILEN = D%NLEN_M(JW,1,JM)*IFIELD
    IF( ILEN > 0 )THEN
      IOFFS = (D%NSTAGT0B(JW)+D%NOFF_M(JW,1,JM))*IFIELD
      IOFFR = (D%NSTAGT0BW(JW,MYSETW)+D%NOFF_M(JW,1,JM))*IFIELD
      FOUBUF_C(IOFFR+1:IOFFR+ILEN)[IPE]=FOUBUF_IN(IOFFS+1:IOFFS+ILEN)
    ENDIF
    ILENS = D%NLEN_M(JW,2,JM)*IFIELD
    IF( ILENS > 0 )THEN
      IOFFS = (D%NSTAGT0B(JW)+D%NOFF_M(JW,2,JM))*IFIELD
      IOFFR = (D%NSTAGT0BW(JW,MYSETW)+D%NOFF_M(JW,2,JM))*IFIELD
      FOUBUF_C(IOFFR+1:IOFFR+ILENS)[IPE]=FOUBUF_IN(IOFFS+1:IOFFS+ILENS)
    ENDIF
  ENDDO
ENDDO
!$OMP END PARALLEL DO
SYNC IMAGES(D%NMYSETW)
FOUBUF(1:IBLEN)=FOUBUF_C(1:IBLEN)[MYPROC]
```

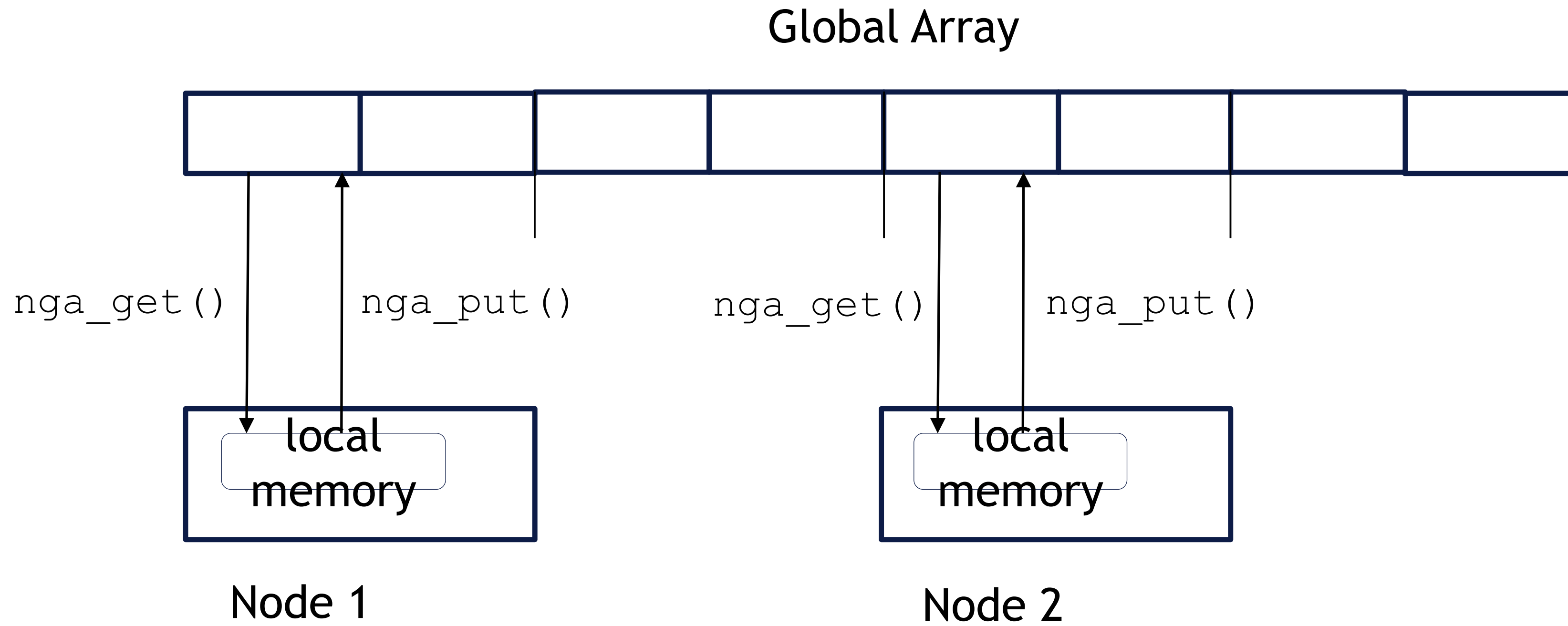


[1] “A PGAS Implementation of the ECMWF Integrated Forecasting System (IFS) NWP Model”, George Mozdzynski

Global Arrays (1)

- PGAS programming model;
- *Shared* and *distributed* memory modes (compile time dependent)
- Interoperable with MPI;
- Use the `NGA_CREATE( )` subroutine to create a global array;
- Use `NGA_PUT( )` and `NGA_GET( )` subroutines to get and put memory from global array into local memory and vice versa;
- A collection of collective subroutines;
- Only has FORTRAN 77 bindings and code must be preprocessed as header files are required using the `#include` directive.

Global Arrays (2)



Global Arrays (3)

```
01    use mpi
02    implicit none
03 #include "mafdecls.fh"
04 #include "global.fh"
05    call mpi_init( ierr )
06    call ga_initialize()
07    nProcs = ga_nNodes()
08    procN = ga_nodeId()
09    print *, “I am process”, procN, “of”, nProcs
10    call ga_terminate()
11    call mpi_finalize( ierr )
```

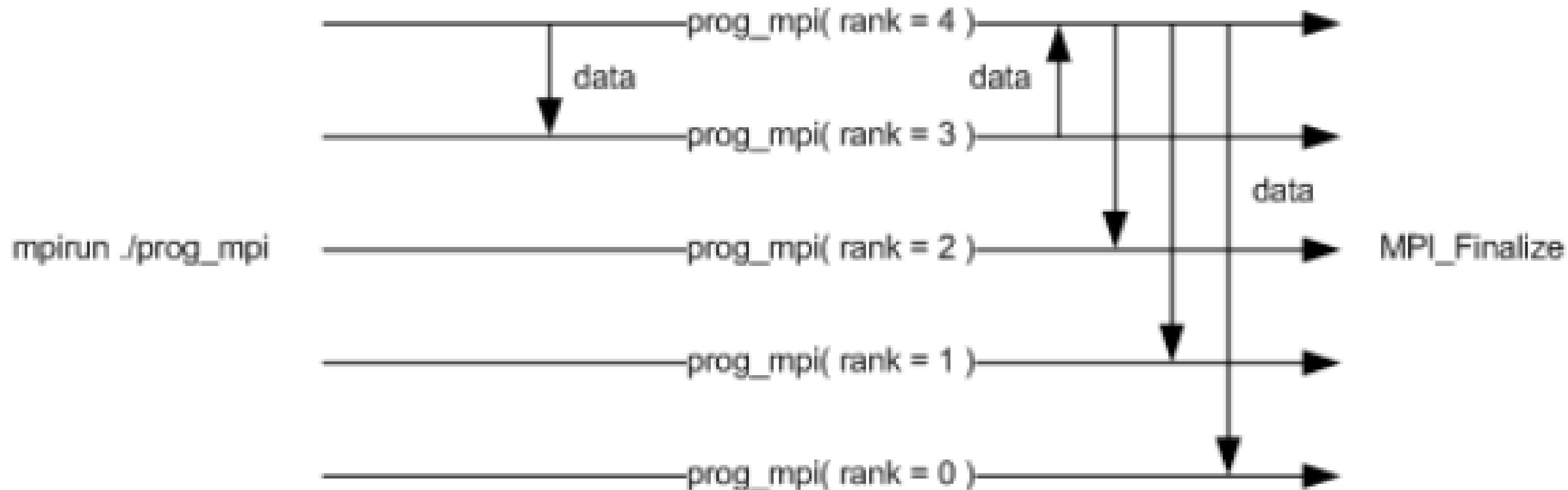
MPI (1)

- The Message Passing Interface, is a standardised and portable message passing specification for *distributed memory* systems;
- It spawns processes which are finalised when program ends;
- Processes can communicate *point-to-point*: a single sending process and a single receiving process
- *One-to-many*: a single sending process and multiple receiving processes
- *Many-to-one*: many sending processes and one receiving process
- *Many-to-many*: multiple sending processes and multiple receiving processes

www.mpi-forum.org

MPI (2)

- Each process is also called a rank and has its own memory space;
- A process must explicitly communicate with another process;
- “More complicated” than OpenMP, Coarray and Global Arrays



nag

MPI (3)

```
01 use mpi
02 real :: a_P
03 real, allocatable :: X_P(:), Y_P(:), Z_P(:)
04 integer :: n_P
05 integer :: nProcs, procN, err
06   call mpi_init(err)
07   call mpi_comm_size(mpi_comm_world, nProcs, err)
08   call mpi_comm_rank(mpi_comm_world, procN, err)
09   n_P = n / nProcs
10   allocate(X_P(n_P)); allocate(Y_P(n_P)); allocate(Z_P(n_P))
11   call mpi_bcast(a_P, 1, mpi_real, 0, mpi_comm_world, err)
12   call mpi_scatter(X, n_P, mpi_real, X_P, n_P, &
                      mpi_real, 0, mpi_comm_world, err)
13   call mpi_scatter(Y, n_P, mpi_real, Y_P, n_P, &
                      mpi_real, 0, mpi_comm_world, err)
14   call axpy(n_P, a_P, X_P, Y_P, Z_P)
15   call mpi_gather(Z_P, n_P, mpi_real, Z, n_P, &
                     mpi_real, 0, mpi_comm_world, err)
16   call mpi_finalize(err)
```

OpenACC (1)

- PGI and Cray compilers fully support OpenACC for Fortran and partial support from GNU Fortran;
- It is similar to OpenMP in that the developer annotates their code for execution on the GPU, thus is much simpler than CUDA Fortran;
- Supports both Nvidia and AMD GPUs.

www.openacc.org
devblogs.nvidia.com/parallelforall/

OpenACC (2)

```
01 !$acc kernels
02   do i = 1, n
03     Z(i) = a * X(i) + Y(i)
04   end do
05 !$acc end kernels
```

CUDA Fortran (1)

- CUDA Fortran is the Fortran version of CUDA C and is only supported by the PGI [1] and IBM compilers;
- CUDA provides a low level interface to Nvidia GPU cards is more difficult than OpenACC but provides more flexibility and opportunities for optimization;
- CUDA Fortran provides language *extensions* and *are not part of the Fortran standard*;
- Example CUDA Fortran codes for materials scientists can be found at [2];

[1] www.pgroup.com [2] https://github.com/RSE-Cambridge/2017-cuda-fortran-material

CUDA Fortran (2)

```
01 attributes(global) subroutine axpy(n, a, X, Y, Z)
02 integer, value :: n
03 real, value :: a
04   i = threadIdx%x + (blockIdx%x - 1) * blockDim%x
05   if (i <= n) Z(i) = a * X(i) + Y(i)

06 use cudafor
07 real, allocatable, device :: X_D(:), Y_D(:), Z_D(:)
08 type(dim3) :: block, grid
09   allocate(X_D(n)); allocate(Y_D(n)); allocate(Z_D(n))
10   err = cudaMemCpy(X_D, X, n, cudaMemCpyHostToDevice)
11   err = cudaMemCpy(Y_D, Y, n)
12   block = dim3(128, 1, 1); grid = dim3(n / block%x, 1, 1)
13   call axpy<<<grid, block>>>(%val(n), %val(a), X_D, Y_D, Z_D)
14   Z(:) = Z_D(:)
```

CUDA Fortran for DO Loops (1)

- You can only operate on device memory:

```
01 !$cuf kernel do(2) <<< *, * >>>
02 do j=1, ny
03  do i = 1, nx
04     a_d(i, j) = b_d(i, j) + c_d(i, j)
05  end do
06 end do
```

CUDA Fortran for DO Loops (2)

- Reduction is automatically generated:

```
01 rsum = 0.0
02 !$cuf kernel do <<<*, *>>>
03 do i = 1, nx
04  rsum = rsum + a_d(i)
05 end do
```

Vectorization (1)

- Parallelism *within single CPU core;*
- Executes Single Instruction on Multiple Data (SIMD);
- General advice is to *let the compiler do the work* for you;
- Fortran array operations usually vectorised by compiler (check compiler feedback);
- If compiler is unable to vectorise and you know it is safe to do so, you can force vectorisation.

www.nersc.gov/users/computational-systems/edison/programming/vectorization/

Vectorization (2)

```
01    do i = 1, n
02      Z(i) = a * X(i) + Y(i)
03    end do
```

```
01    Z(1:n) = a * X(1:n) + Y(1:n)
```

```
01    do i = 1, n, 4
02      Z(i)   = a * X(i)   + Y(i)
03      Z(i+1) = a * X(i+1) + Y(i+1)
04      Z(i+2) = a * X(i+2) + Y(i+2)
05      Z(i+3) = a * X(i+3) + Y(i+3)
06    end do
```

Vectorization (3)

```
01    do i = 1, n, 4
02      <load X(i), X(i+1), X(i+2), X(i+3) into X_v>
03      <load Y(i), Y(i+1), Y(i+2), Y(i+3) into Y_v>
04      Z_v = a * X_v + Y_v
05      <store Z_v into Z(i), Z(i+1), Z(i+2), Z(i+3)>
06    end do
```

```
01    do i = 1, n, 4
02      Z(i)   = a * X(i)   + Y(i)
03      Z(i+1) = a * X(i+1) + Y(i+1)
04      Z(i+2) = a * X(i+2) + Y(i+2)
05      Z(i+3) = a * X(i+3) + Y(i+3)
06    end do
```

Vectorization (4)

```
01 !$omp simd
02   do i = 1, n
03     Z(i) = a * X(i) + Y(i)
04   end do
05 !$omp end simd
```

nag®

Summary of Parallel Models (1)

- OpenMP is easy to use and test. OpenMP can be switched on/off simply by using a compiler flag;
- However, OpenMP is limited to a single memory space node and can suffer limited scalability. Race conditions can also occur which are difficult to debug;
- MPI offers higher scalability and can run on multiple server nodes, thus offering larger memory space;
- However, MPI is more difficult to use and requires code rewrite if parallelising sequential code;

Summary of Each Parallel Models (2)

- MPI can suffer from process deadlocks, and race conditions when doing parallel I/O;
- CoArray is simple to use and runs across a number of server nodes. Easy to partition data across images.
- However, race conditions can easily occur and the developer is solely responsible for preventing them;
- Global Arrays is simpler to use than MPI and easy to partition data across processes. It is fully compatible with MPI;
- Like CoArray, race conditions can easily occur and must be managed.

Summary of Each Parallel Models (3)

- PGAS parallel models are beneficial if you have irregular communication in your code. Implementing irregular communication in MPI is more difficult;
- CoArray offers convenient way to parallelise code using language syntax and is also beneficial for irregular communication;
- Performance of different parallel models, e.g. MPI and coarray, will vary depending on how the runtime system is implemented. You must benchmark to determine performance;
- One-sided communication in MPI is more difficult.

Summary of Each Parallel Models (3)

- GPUs can provide high levels of performance assuming that your code is highly parallelisable and need not be cache efficient;
- High memory bandwidth of GPU memory;
- Multi-GPU executions on a single server node and multiple server nodes;
- Disadvantage is that codes must have high levels of parallelism;
- High memory latency between the CPU and GPU, thus a large amount of data must be moved to the GPU in a single transfer;
- The GPU must do sufficient amount of computation to offset cost of data transfer.

Parallel I/O using HDF5 and NetCDF

Parallel I/O

- I/O is often the most under-considered part of a program.
- At the end of a job, data needs to be stored for follow-on runs or post-processing.
  - The time spent doing I/O is often ignored as a “necessary evil”.
  - It can in reality be very expensive.
  - It may also be repeated at various points in the code execution, making its effect more significant.
- Parallel I/O aims to allow the user to read and write from a single file using any number of processes.

Parallel I/O

- On parallel machines, I/O can become a major bottleneck.
- Ken Batcher, professor of computer science at Kent State University, coined the definition that
  - "A supercomputer is a device for turning compute-bound problems into I/O bound problems."
- Any code which uses a single process to perform I/O on behalf of all processes will serialise that part of the application.
- Running the I/O in parallel on a parallel file system should allow potential benefits in terms of the scalability of the whole code.
  - If the I/O is not parallelised, it will hit scalability problems, as shown by Amdahl's Law.
  - A parallel I/O file system is required for much improvement in I/O throughput.

Common I/O Strategies

- Use one process to do all input and output.
  - Collects data from other processes and outputs to disk.
  - Serialises the output.
- Each process outputs one file.
  - Parallelises output.
  - Limits ability to change number of processes.
- Combination of above two strategies.
  - One process outputs data on behalf of a few processes.

Parallel I/O Libraries

- There are several I/O libraries which can enable parallel access to files.
- Two popular libraries, presented yesterday, are NetCDF and HDF5.
  - NetCDF is Networked Common Data Format and HDF5 stands for Hierarchical Data Format (version 5).
  - Both allow architecture neutral files to be created.
  - Both have parallel versions of the serial libraries.
- The parallel features of these libraries are built on top of MPI-IO.
  - You need to know aspects of MPI-IO to effectively use these parallel libraries.

Parallel HDF

- Parallel HDF5 (PHDF5) is a library for parallel I/O in the HDF format.
- It has C and Fortran interfaces.
- The files are compatible with serial HDF5 files and sharable between serial and parallel platforms.
- PHDF5 is designed to have a single file image to all processes.
- PHDF5 supports MPI programming but not shared memory programming. It is built on MPI-IO.

Parallel HDF

- In PHDF5 you open a parallel file within an MPI communicator.
- All processes are required to participate. Most of the PHDF5 calls in the API are collective. Different files can be opened using different communicators.
- With the Parallel HDF5 collective API you can create, open and close objects.
- Reading and writing to datasets can be done non-collectively.
- Once a file is opened by the processes of a communicator:
  - All parts of the file and objects are accessible by all processes.
  - Multiple processes can write to the same dataset, or their own.

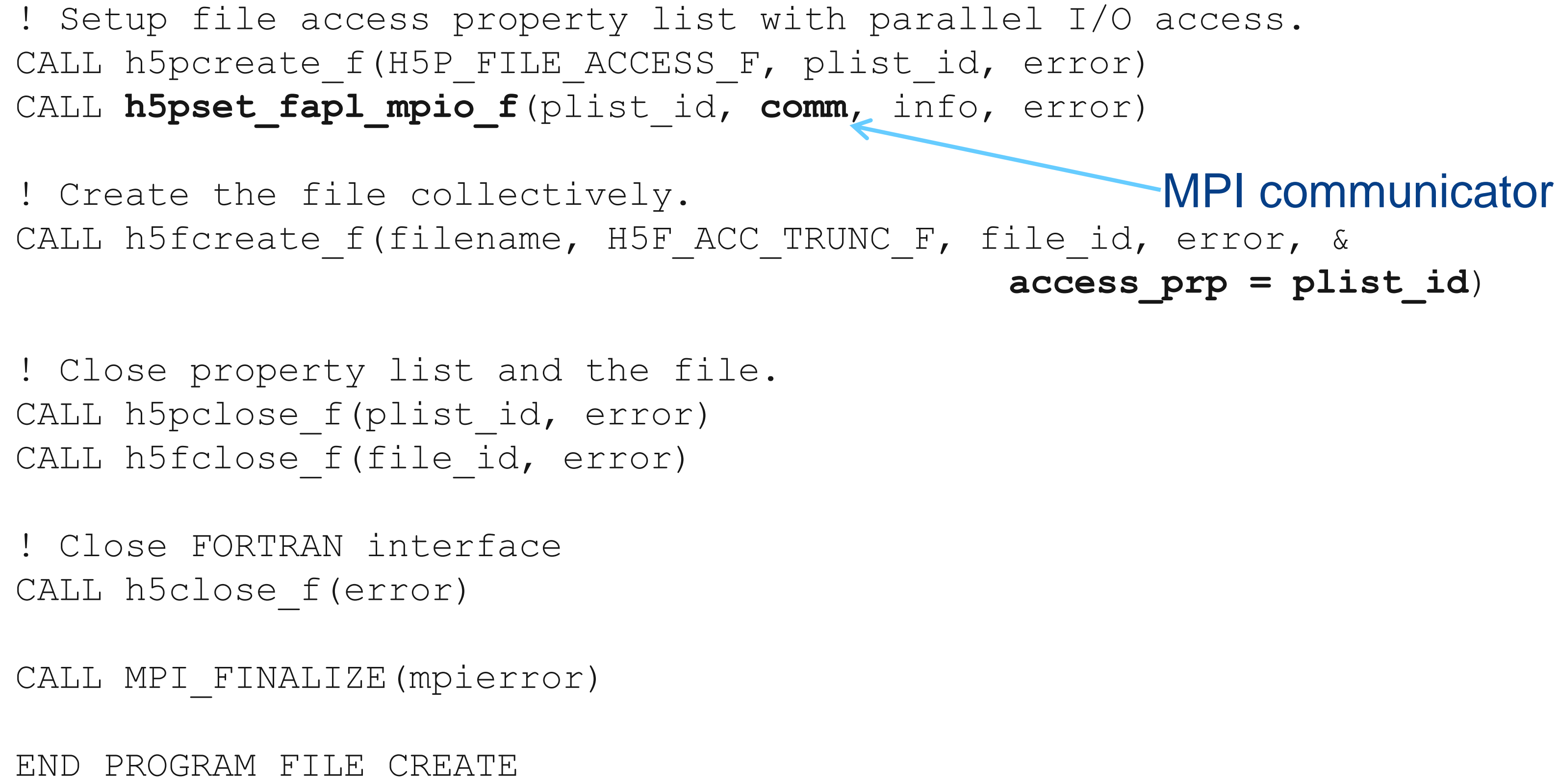
Creating and Accessing Files

- The programming model for creating and accessing a file is as follows:
  1. Set up an access template object to control the file access mechanism.
  2. Open the file.
  3. Close the file.
- Each process of the MPI communicator creates an access template. This is done with the H5Pcreate / _f call to obtain the file access property list and the H5Pset_fapl_mpio / _f call to set up parallel I/O access.
- An example code follows for creating an access template in PHDF5:

Parallel HDF Fortran Example

```
PROGRAM FILE_CREATE

    USE HDF5 ! This module contains all necessary modules
    USE MPI
    IMPLICIT NONE

    CHARACTER(LEN=10), PARAMETER :: filename = "sds.h5"  ! File name

    INTEGER(HID_T) :: file_id       ! File identifier
    INTEGER(HID_T) :: plist_id      ! Property list identifier
    INTEGER        :: error
    INTEGER :: mpierror       ! MPI error flag
    INTEGER :: comm, info
    INTEGER :: mpi_size, mpi_rank
    comm = MPI_COMM_WORLD
    info = MPI_INFO_NULL

    CALL MPI_INIT(mpierror)
    CALL MPI_COMM_SIZE(comm, mpi_size, mpierror)
    CALL MPI_COMM_RANK(comm, mpi_rank, mpierror)

    ! Initialize FORTRAN predefined datatypes
    CALL h5open_f(error)
```

Parallel HDF Fortran Example




```
! Setup file access property list with parallel I/O access.
CALL h5pcreate_f(H5P_FILE_ACCESS_F, plist_id, error)
CALL h5pset_fapl_mpio_f(plist_id, comm, info, error)

! Create the file collectively.
CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id, error, &
                                             access_prp = plist_id)


! Close property list and the file.
CALL h5pclose_f(plist_id, error)
CALL h5fclose_f(file_id, error)

! Close FORTRAN interface
CALL h5close_f(error)

CALL MPI_FINALIZE(mpierror)

END PROGRAM FILE_CREATE
```

Creating and Accessing Datasets

- The programming model for accessing a dataset with Parallel HDF5 is:

  1. Set up file access property list with parallel I/O access
  2. Create a new file collectively
  3. Create the dataspace for the dataset
  4. Create the dataset
  5. Create property list for the parallel dataset write
  6. Write to the dataset
  7. Close and release any resources

- For example:

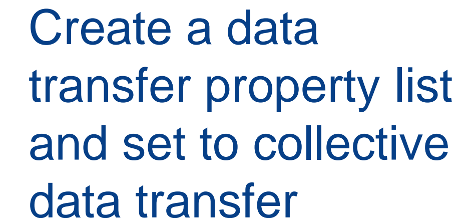
nag

PHDF5 Dataset Example Fortran

```
! Initialize FORTRAN interface
CALL h5open_f(error)

! Setup file access property list with parallel I/O access.
CALL h5pcreate_f(H5P_FILE_ACCESS_F, plist_id, error)
CALL h5pset_fapl_mpio_f(plist_id, comm, info, error)

! Create the file collectively.
CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id, error, access_prp = plist_id)
CALL h5pclose_f(plist_id, error)

! Create the data space for the dataset.
CALL h5screate_simple_f(rank, dimsf, filespace, error)

! Create the dataset with default properties.
CALL h5dcreate_f(file_id, dsetname, H5T_NATIVE_INTEGER, filespace, &
                 dset_id, error)
```

PHDF5 Dataset Example Fortran

```
! Create property list for collective dataset write
CALL h5pcreate_f(H5P_DATASET_XFER_F, plist_id, error)
CALL h5pset_dxpl_mpio_f(plist_id, H5FD_MPIO_COLLECTIVE_F, error)
! For independent write use H5FD_MPIO_INDEPENDENT_F

! Write the dataset collectively.
CALL h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, data, dimsfi, error, &
                xfer_prp = plist_id)

! Deallocate data buffer.
DEALLOCATE(data)

! Close resources.
CALL h5sclose_f(filespace, error)
CALL h5dclose_f(dset_id, error)
CALL h5pclose_f(plist_id, error)
CALL h5fclose_f(file_id, error)

! Close FORTRAN interface
CALL h5close_f(error)
```



Creating and Accessing Dataset

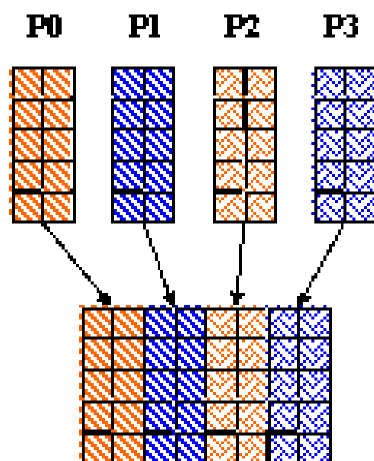
- All processes that have opened a dataset may do collective I/O.
- Each process may do an independent and arbitrary number of data I/O access calls by setting h5pset_dxpl_mpio / _F appropriately.
- If a dataset is unlimited, you can extend it with a *collective call* to H5Dextend / h5dextend_f

Writing Hyperslabs

- Each process defines the memory and file hyperslabs.
- Each process executes a partial write/read call which is either collective or independent.
- The memory and file hyperslabs in the first step are defined with the H5Sselect_hyperslab / h5sselect_hyperslab_f.
- The *start* (or *offset*), *count*, *stride* and *block* parameters define the portion of the dataset to write to.
- By changing the values of these parameters you can write hyperslabs by contiguous hyperslab, regularly spaced data in a column/row, pattern or chunk.

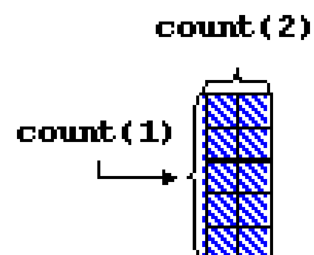
Writing Hyperslabs

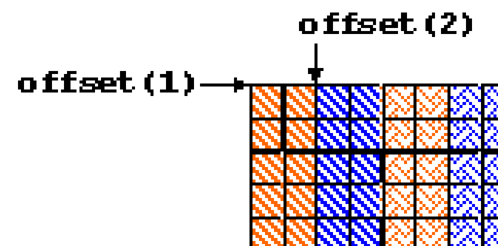
- Contiguous hyperslab in Fortran



P1 (memory space)

File

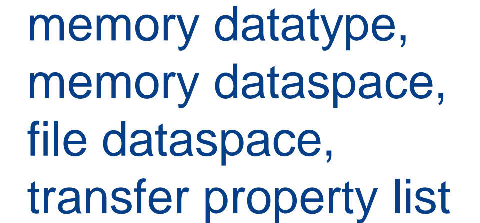





```
count(1)  = dimsf(1)
count(2)  = dimsf(2)/mpi_size
offset(1) = 0
offset(2) = mpi_rank * count(2)
```

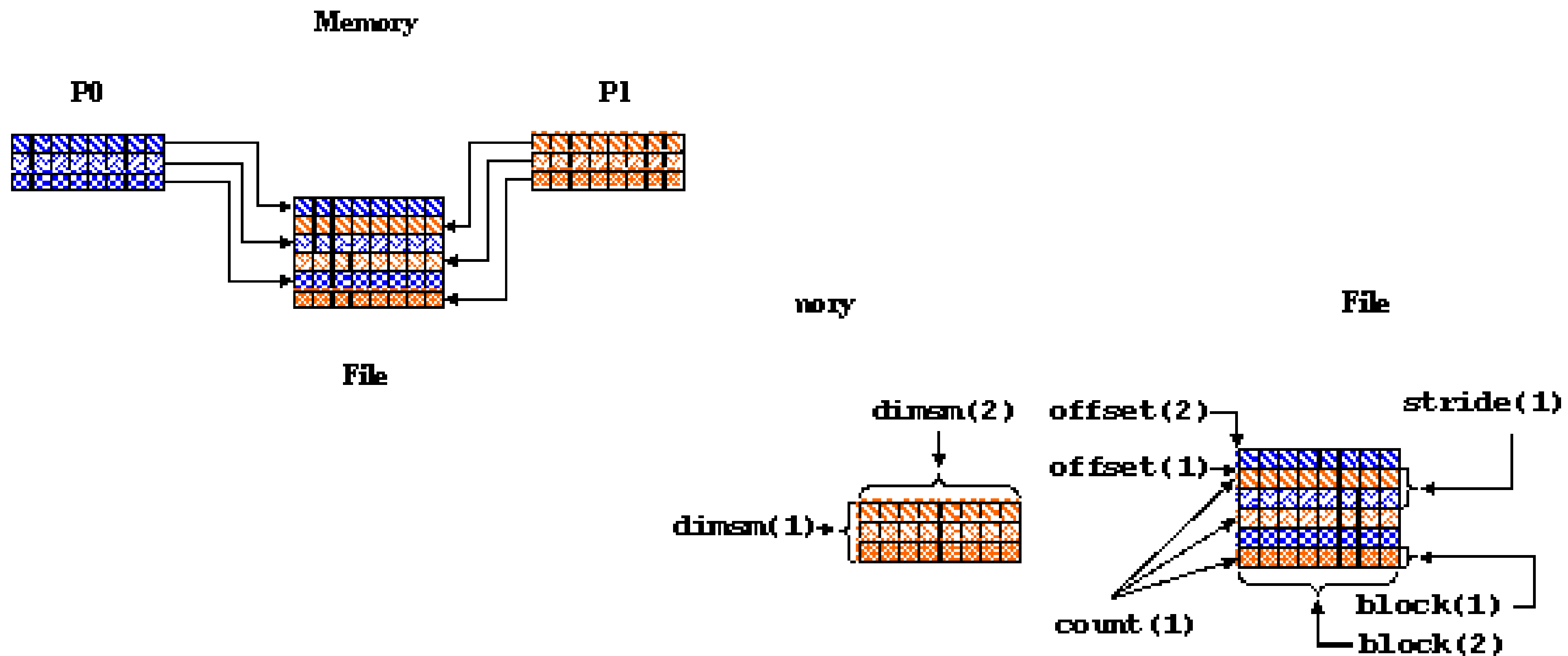
Writing Hyperslabs

```
! Each process defines dataset in memory and writes it to the hyperslab
! in the file.
count(1) = dimsf(1)
count(2) = dimsf(2)/mpi_size
offset(1) = 0
offset(2) = mpi_rank * count(2)
CALL h5screate_simple_f(rank, count, memspace, error)

! Select hyperslab in the file.
CALL h5dget_space_f(dset_id, filespace, error)
CALL h5sselect_hyperslab_f (filespace, H5S_SELECT_SET_F, offset, count, error)

! Create property list for collective dataset write
CALL h5pcreate_f(H5P_DATASET_XFER_F, plist_id, error)
CALL h5pset_dxpl_mpio_f(plist_id, H5FD_MPIO_COLLECTIVE_F, error)

! Write the dataset collectively.
CALL h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, data, dimsfi, error, &
                file_space_id = filespace, mem_space_id = memspace, xfer_prp = plist_id)
```



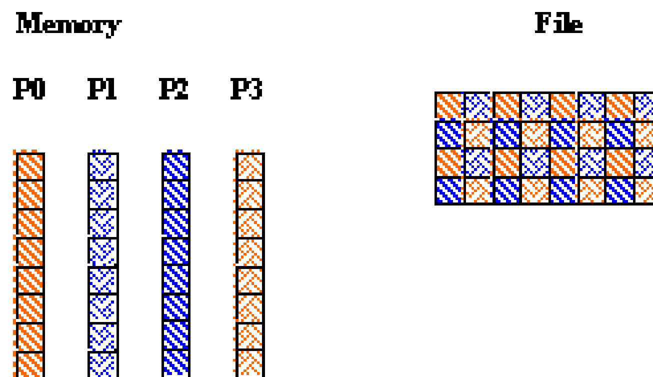
Writing Hyperslabs

- Regularly spaced data in Fortran

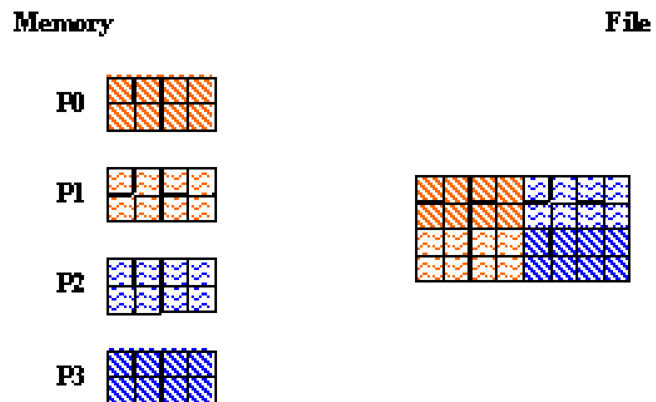


Writing Hyperslabs

- By pattern



- By chunk



Parallel I/O with NetCDF

- For parallel I/O, we require NetCDF-4.
- NetCDF-4 uses the parallel I/O features of HDF5.
  - Allowing many processes to read/write NetCDF data at the same time.
  - This requires an implementation of MPI-2.
    - MPICH and OpenMPI are free implementations that can be used and supercomputers often have a proprietary implementation of MPI-2.
  - Other methods of parallelism are not supported by NetCDF.

Some History: The pNetCDF Package

- The pnetcdf package from Argonne and Northwestern can be used for parallel I/O with classic netCDF data (i.e. pre-NetCDF-4).
  - For classic and 64-bit offset formats, parallel I/O can be obtained with pnetcdf, the parallel netCDF package from Argonne and Northwestern.
  - pnetCDF is well-tested and maintained. See https://parallel-netcdf.github.io/ for info.
  - pNetCDF uses MPI-IO to perform parallel I/O. It is a complete rewrite of the core C library using MPI-IO instead of POSIX.
  - Unfortunately, the pnetCDF package implements a different API from the netCDF API, making portability with other netCDF codes a problem.
  - Probably best to use NetCDF-4 nowadays.

Opening NetCDF-4 Files in Parallel

- In Fortran
  - Simply provide the optional parameters `comm` and `info` for nf90_create or nf90_open.

```
mode_flag = nf90_netcdf4
call handle_err(nf90_create(FILE_NAME, mode_flag, ncid, &
                comm = MPI_COMM_WORLD, info = MPI_INFO_NULL))
```

Collective and Independent Operations

- **netCDF operations may be collective (must be done by all processes at the same time) or independent (can be done by any process at any time).**
  - All netCDF metadata writing operations are collective. That is, all creation of groups, types, variables, dimensions or attributes.
  - Data reads and writes may be independent (the default) or collective. To make writes to a variable collective, call the nf90_var_par_access function.

Parallel I/O Example

```
stat = NF90_CREATE(FILE_NAME, NF90_NETCDF4, ncid, comm=MPI_COMM_WORLD, &
  info=MPI_INFO_NULL)

stat = NF90_DEF_DIM(ncid, "x", np, x_dimid)
stat = NF90_DEF_DIM(ncid, "y", np, y_dimid)
dimids = [ y_dimid, x_dimid ]
stat = NF90_DEF_VAR(ncid, "data", NF90_INT, dimids, varid)
stat = NF90_ENDDEF(ncid)

starts = [ 1, my_rank+1 ]
counts = [ np, 1 ]
stat = NF90_PUT_VAR(ncid, varid, data_out, start=starts, count=counts)

stat = NF90_CLOSE(ncid)
```

Fortran Interoperability with C

Fortran Interoperability with C

- C is another major programming language in computational science and Fortran 2003 provides an interface to it;
- It uses the `iso_c_binding` intrinsic Fortran module;
- If passing two-dimensional arrays between C and Fortran, remember to transpose the array;
- **Only assumed sized arrays are supported in 2008. Assumed shaped arrays are only supported in Fortran 2018**;

Fortran Kind Type	Equivalent C Type
`C_INT`	`int`
`C_FLOAT`	`float`
`C_DOUBLE`	`double`

nag

Calling Fortran from C (1)

```
/* sum_c.c */
#include <stdio.h>

float sum_f( float *, int * );
int main( int argc, char *argv[] ) {
  float x[4] = { 1.0, 2.0, 3.0, 4.0 };
  int n = 4;
  float res;

  res = sum_f( x, &n );
}
```

```
! sum_f.f90
function sum_f( x, n ) result ( res ) &
  bind( C, name = 'sum_f' )
  use iso_c_binding
  implicit none

  real(kind=C_FLOAT), intent(in) :: x(*)
  integer(kind=C_INT), intent(in) :: n
  real(kind=C_FLOAT) :: res

  res = sum( x(1:n) )
end function sum_f
```

Calling Fortran from C (2)

- Compile both files:

```
$ gfortran -c sum_f.f90
$ gcc -c sum_c.c
```

- The `bind` attribute removes the leading underscore in the symbol table:

```
$ nm sum_f.o
0000000000000000 T sum_f
```

- Then do the final link - object files must be listed in this order:

```
$ gcc sum_c.o sum_f.o -o sum_c.exe
```

Calling C from Fortran (1)

```
! sum_f.f90
program sum_f
  use iso_c_binding
  interface
    function sum_c( x, n ) bind( C, name = 'sum_c' )
      use iso_c_binding
      real(kind=C_FLOAT) :: sum_c
      real(kind=C_FLOAT) :: x(*)
      integer(kind=C_INT), value :: n
    end function sum_c
  end interface
  integer(kind=C_INT), parameter :: n = 4
  real(kind=C_FLOAT) :: x(n) = [ 1.0, 2.0, 3.0, 4.0 ]
  print *, sum_c( x, n )
end program sum_f
```

```
/* sum_c.c */
float sum_c( float *x, int n )
{
  float sum = 0.0f;
  int i;

  for ( i = 0; i < n; i++ ) {
    sum = sum + x[i];
  }

  return sum;
}
```

Calling C from Fortran (2)

- Compile both files:

```
$ gcc -c sum_c.c
$ gfortran -c sum_f.f90
```

- The `bind` attribute tells the interface to call the function `sum_c` which is listed in the symbol table:

```
$ nm sum_c.o
0000000000000000 T sum_c
```

- Then do the final link - object files must be listed in this order:

```
$ gfortran sum_f.o sum_c.o -o sum_f.exe
```

Fortran 2018 Interoperability with C

- Optional dummy arguments - `optional` attribute;
- Assumed-length character dummy arguments - `character(len=*), intent(in) :: header`
- Assumed shaped arrays - `real, intent(in) :: vec(:)`
- Allocatable dummy arguments - `real, allocatable, intent(out) :: table(:, :)`
- Pointer dummy arguments - `real, pointer, intent(in) :: vec(:)`

Optional Dummy Arguments (1)

- The optional argument is passed as a pointer to C. If the dummy argument is a NULL pointer, then it is not present;

```
subroutine print_header( debug )
  use iso_c_binding
  integer(C_INT), optional :: debug
  if ( present( debug )) then
    print '(I0,1X,A)', debug, 'Error found'
  else
    print '(1X,A)', 'Error found'
  end if
end subroutine
```

Optional Dummy Arguments (2)

- To call **with** the optional argument in the C code:

```
int debug = 4;
print_header( &debug );
```

- To call **without** the optional argument:

```
print_header ( (int *)0 );
```

Assumed-Length Character Dummy Arguments (1)

- Fortran calling C print function using descriptors:

```
interface
  subroutine print_header( msg ) bind(C)
    use iso_c_binding
    character(len=*,kind=c_char) , intent(in) :: msg
  end subroutine print_header
end interface
```

nag

Assumed-Length Character Dummy Arguments (2)

```
#include <stdio.h>
#include “iso_fortran_binding.h”
void print_header( CFI_cdesc_t *msg ) {
  int ind;
  char *p = msg->base_addr;
  for ( ind = 0; ind < msg->elem_len; ind++ )
    putc( p[ind], stdout );
  putc( ‘\n’, stdout );
}
```

nag®

C Descriptors (1)

- A C descriptor (`CFI_cdesc_t`) is a C structure with the following members:

`void *base_addr` - the address of the object. For unallocatable or disassociated pointers, it is NULL;

`size_t elem_len` - storage size in bytes;

`int version` - version number of the descriptor;

`CFI_attribute_t attribute` - whether the object is allocatable (`CFI_attribute_allocatable`), pointer (`CFI_attribute_pointer`) or neither (`CFI_attribute_other`).

`CFI_rank_t rank` - rank of the object and zero if a scalar;

C Descriptors (2)

`CFI_type_t type` - data type of this object. Macro can be `CFI_type_int`, `CFI_type_float`, `CFI_type_double`, `CFI_double_Complex`, and many other macros;

`CFI_dim_t dim[]` - describing the shape, bounds and memory layout of the array object;

- `CFI_index_t lower_bound` - the lower bound of array. Zero for everything else (member of `dim`);
- `CFI_index_t extent` - size of the dimension (member of `dim`);
- `CFI_index_t sm` - memory stride (member of `dim`).

nag®

C Example

```
void abs_array( CFI_cdesc_t *array )
  size_t i, nel = 1;
  for ( i = 0; i < array->rank; i++)
     nel = nel * array->dim[i].extent;

  if ( array->type == CFI_type_float ) {
    float *f = array->base_addr;
    for ( i = 0; i < nel; i++) f[i] = fabs( f[i] );
  } /* and for other real types */
}
```

Fortran Interoperability with Python

- Fortran subroutines and functions can be called from Python;
- Take advantage of the speed of Fortran with the ease of Python;
- Computationally intensive functions are implemented in Fortran to provide the speed and efficiency;
- Python is a widely supported scripting language with a huge number of well supported libraries, e.g. NumPy, SciPy, Matplotlib;
- *Extend the concept of reusable code to other programming languages*;
- Python already calls many Fortran subroutines, e.g. in BLAS and LAPACK is called in SciPy.

Example Fortran Module

```
module sum_mod
contains
  subroutine sumpy( array_f, result_f )
     real, dimension(:), intent(in) :: array_f
     real, intent(out) :: result_f
     result_f = sum( array_f )
   end subroutine sumpy
   function fumpy( array_f ) result( result_f )
     real, dimension(:), intent(in) :: array_f
     real :: result_f
     result_f = sum( array_f )
     end function fumpy
end module sum_mod
```

Calling Fortran from Python

- To compile the previous example:

```
$ f2py -c --fcompiler=gnu95 -m sum_mod sum_mod.F90
```

- For list of other supported compilers:

```
$ f2py -c --help-fcompiler
```

- Will create the shared object library `sum_mod.so` which is *imported*:

```
from sum_mod import sum_mod;
import numpy;
a = sum_mod.sumpy( [ 1.0, 2.0 ] );
b = sum_mod.fumpy( [ 1.0, 2.0 ] );
c = sum_mod.sumpy( numpy.array( [ 1.0, 2.0 ] ) );
```

- The F90WRAP [1] tool is a better tool for calling Fortran from Python.

[1] https://github.com/jameskermode/f90wrap

nag

Fortran Interoperability with R (1)

- The statistical language R can only use Fortran subroutines;

```
module sums_mod
contains
subroutine rsum( array_f, len, result_f ) &
                 bind(C, name = "sums_mod_rsum_")
  integer, intent(in) :: len
  real(kind=DP), dimension(0:len - 1),intent(in) :: array_f
  real(kind=DP), intent(out) :: result_f

  result_f = sum( array_f(0:len - 1) )
end subroutine rsum

end module sums_mod
```

Fortran Interoperability with R (2)

- Build a dynamic library (shared object):

```
$ gfortran -c sums_mod.F90
$ gfortran -shared sums_mod.o -o sums_mod.so
```

- Then load it in R:

```
> dyn.load( “sums_mod.so” )
> .Fortran( “sums_mod_rsum”, array_f = as.double( 1:4 ),
            len = length( 1:4 ), c = as.double( 0 ))
$array_f
[1] 1 2 3 4
$len
[1] 4
$c
[1] 10
```

Why do you need Numerical Libraries?

Don‘t try to reinvent the wheel

- Would you reimplement tools like Git, Valgrind, etc.?
- Are you paid for writing numerical components?
- Numerical Library is a tool and a building block to help you develop your scientific applications.

Why do you need Numerical Libraries?

- Model your code at a higher level
  - Use numerical algorithm instead of implementing them (e.g. optimizers)
- Concentrate on your task and core expertise
- Reduce development time
- Reduce maintenance time
- Use external expertise
  - choice of algorithms, support, speed

Desired Properties of Numerical Libraries

- Reliable components
  - Stable numerical algorithms
- Coverage and availability of alternative algorithms
- Portability
  - Different operating systems, languages, etc.
- Maintenance
  - Bugfixes and regular updates
- Documentation and support
- Parallelization (OpenMP shared memory)

The NAG Library

- Hundreds of routines devoted to numerical analysis and statistics, the NAG Library helps users build applications for many different industries and fields.
- For your current and future programming environments
  - NAG Library routines are available for Fortran, C, C++, Python, .NET, Java, MATLAB and others
  - NAG Library routines can be called many computer languages/environments such as Visual Basic, Octave, Scilab, R, etc.
  - Assists migration of applications to different environments

NAG Library Full Contents

- Root Finding
- Summation of Series
- Quadrature
- Ordinary Differential Equations
- Partial Differential Equations
- Numerical Differentiation
- Integral Equations
- Mesh Generation
- Interpolation
- Curve and Surface Fitting
- Optimization
  - Approximations of Special Functions

- Dense Linear Algebra
- Sparse Linear Algebra
- Correlation & Regression Analysis
- Multivariate Methods
- Analysis of Variance
- Random Number Generators
- Univariate Estimation
- Nonparametric Statistics
- Smoothing in Statistics
- Contingency Table Analysis
- Survival Analysis
- Time Series Analysis
- Operations Research

Why use NAG Libraries and Toolboxes?

- Global reputation for quality – accuracy, reliability and robustness…
- Extensively tested, supported and maintained code
- Reduces development time
- Allows concentration on your key areas
- Components
  - Fit into your environment
  - Simple interfaces to your favourite packages
- Regular performance improvements!
- Give “qualified error” messages e.g. tolerances of answers

NAG Library - Ease of integration

- Supporting Wide Range of Operating systems…
  - Windows, Linux, Mac, …
- and a number of interfaces
  - C, C++
  - Fortran
  - VB, Excel & VBA
  - C#, F#, VB.NET
  - Java
  - Python
  - Julia
  - Excel
  - MATLAB
  - Hadoop / Apache Spark
  - LabVIEW
  - R, S-Plus
  - Mathematica
  - Scilab, Octave

NAG development philosophy

- First priority: ***accuracy***
- Second priority: ***performance***
- Algorithms chosen for
  - usefulness
  - robustness
  - accuracy
  - stability
  - speed

NAG Technical Support Service

- Single point of contact: dedicated technical desk
- Highly knowledgeable team
  - Support from the subroutine developers
- Advice on a wide range of areas including
  - functionality
  - diagnosis of user problems
  - work around to assist users ahead of standard updates
  - product availability for specific operating systems
  - advice on the best functionality for your application needs
  - wide range of documentation and technical reports
- Updates and access to new releases

NAG Technology Innovation

- Long history of collaboration with the world's leading scientists and engineers across academia, government research and industry
- Examples of ongoing collaboration are:
  - work with mathematicians and statisticians across the globe to produce the best / most competitive algorithms for the NAG Library and bespoke solutions;
  - in accelerator computing / HPC (many core, GPU,…), working closely with:
    - the main hardware vendors (AMD, ARM, Intel and NVIDA)
    - relevant leading academics (inc. Professors Mike Giles, William Shaw, Nick Higham, Jack Dongarra);
  - innovating by working with RWTH Aachen University to deliver Algorithmic Differentiation solutions (Prof. Uwe Naumann et al).

NAG Library

- The NAG Library is divided into chapters, each devoted to a branch of maths or statistics. Each has a 3-character name and a title, e.g., F03 – Determinants.
- Exceptionally, Chapters H and S have one-character names.
- All routines in the Fortran Library have six-character names, beginning with the characters of the chapter name, e.g. `d01aj`**`f`** (last character stands for Fortran).
- There are also "long names" that aim to be more descriptive.

NAG Library Documentation (1)

- Library has complete documentation
  - Distributed in environment appropriate formats including PDF, HTML and MathML formats
  - Chapter introductions
    - technical background to the area
    - assistance in choosing the appropriate routine
  - Routine Documentation
    - description of method
    - specification of each parameter
    - explanation of error exits
    - example programs
    - remarks on accuracy

NAG Library Documentation (2)

- All documentation is available online
  - https://www.nag.co.uk/numeric/fl/nagdoc_latest/html/frontmatter/manconts.html

First Steps with the NAG Library

- A detailed implementation specific description on how to compile and run the examples is given in User's Note
  - https://www.nag.co.uk/numeric/fl/nagdoc_latest/html/genint/usersnote.html

- The easiest way to start
  - On Windows
    - use `nag_example_*.bat` batch files located in `[INSTALL_DIR]/batch`
    - you might need to run the `envvars.bat` batch file first to set the environment variables
  - On Linux
    - use `nag_example_*` scripts located in `[INSTALL_DIR]/scripts`

Errors and Warnings

- The routine has detected a warning or an error if the value of argument `IFAIL` (or, in chapters F07 and F08 `INFO`) is non-zero on exit
- For details about how to interpret this value the user should consult the Error Indicators and Warnings section of the document for the particular routine

Errors and Warnings

- `IFAIL` argument
  - allow you to specify what action the Library routine should take if an error is detected
  - to inform you of the outcome of the call of the routine
- On input if `IFAIL=`
  - `0:` Hard fail. The execution of the program will terminate if the routine detects an error
  - `1:` Soft fail with silent exit. Returns control to the calling program without output of the error message
  - `-1:` Soft fail with noisy exit. Outputs an error message before the control is returned to the calling program

Don't forget to test the value of `IFAIL` in soft fail mode!

Different Implementations of BLAS and LAPACK

- NAG Fortran Library provides static and shared libraries that use different implementations of BLAS and LAPACK routines
  - Intel MKL (should be used for best performance)
    - Multithreaded
    - `libnag_mkl.a` (Linux), `nag_mkl_M*.lib` (Windows)
    - `libnag_mkl.so` (Linux), `FLW6I26DE_mkl.lib/FLW6I26DE_mkl.dll` (Windows)
  - NAG
    - `libnag_nag.a` (Linux), `nag_nag_M*.lib` (Windows)
    - `libnag_nag.so` (Linux), `FLW6I26DE_nag.lib/FLW6I26DE_nag.dll` (Windows)

Interface blocks and Parallelism

- Interface blocks for all user callable routines
  - Interface blocks are separated by chapters
- Different levels of parallelization
  - Using multithreaded version of Intel MKL
    - BLAS and LAPACK
  - Using NAG Fortran library for SMP & Multicore

NAG Library Interoperability with C/C++ and Python

- Interoperability with C/C++
  - C Headers
  - NAG C library
- Interoperability with Python
  - Full set of bindings available for NAG C library – for Windows, Linux and Mac
  - Access to NAG routines from Python for quick prototyping
  - Same high quality NAG routines used in production system (C, Fortran, .NET, Java, ...) as used under Python prototype
  - Supported by white papers for calling NAG Fortran or C Library from Python
    - https://www.nag.co.uk/nag-library-python

Conclusions – NAG solutions

The NAG Library provides:

- Standard and advanced routines
  - hundreds of numerical routines
- Reliability
  - all routines vigorously tested
  - extensive experience of implementing numerical code
- Portability
  - accessible from many software environments
  - constantly being implemented for new architectures
- Support
  - directly supported by the team that creates the code

End of Workshop - Discussion for 10 minutes

- Have you found this workshop useful?
- What tools, libraries, and techniques will you now use for your code development?
- What aspects of the workshop were not useful?
- What could be improved?
- Do you feel more confident after attending this workshop?

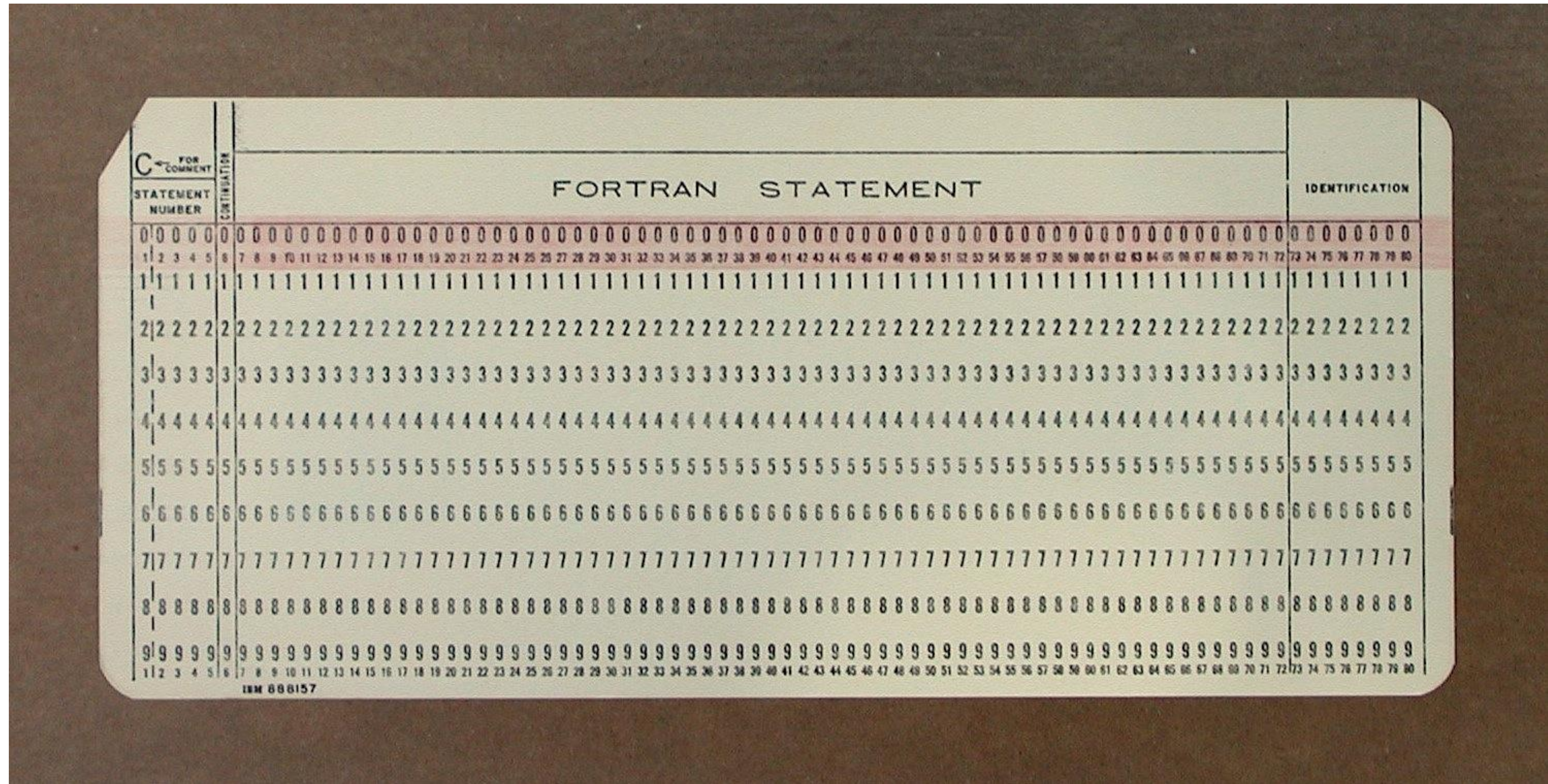
Workshop Feedback

- Please complete workshop feedback at:

`http://www.nag.co.uk/content/fortran-modernization-workshop-feedback`

- For question “where did you attend this workshop?”, please put the name of the city. Many thanks!

End of Day Two - Exercises 2



References (1)

- “Modern Fortran in Practice”, A. Markus. Cambridge University Press, 2012;
- “Modern Fortran”, N. Clerman and W. Spector. Cambridge University Press, 2012;
- “Modern Fortran Explained: Incorporating Fortran 2018”, M. Metcalf, J. Reid and M. Cohen. Oxford University Press, 2018;
- “Git Pocket Guide”, R. Silverman. O’Reilly, 2013;
- "Why Programs Fail", A. Zeller. Morgan Kaufmann, 2009.

References (2)

- “CUDA Fortran for Scientists and Engineers”, G. Ruetsch and M. Fatica. Morgan Kaufmann, 2013;
- “Managing Projects with GNU Make”, R. Mecklenburg. O'Reilly, 2004;
- “Introduction to Programming with Fortran”, I. Chivers and J. Sleightholme. Springer, 2015;
- “Scientific Software Development in Fortran”, Drew McCormack. Lulu, 2010.
- “Numerical Computing with Modern Fortran”, R. Hanson, SIAM. 2014.
- “Guide to Fortran 2008 Programming”, W. Brainerd. Springer. 2015.

References (3)

- "A Guidebook to Fortran on Supercomputers", J. Levesque and J. Williamson. Academic Press, 1989.
- "Programming Models for Parallel Computing", P. Balaji. MIT Press, 2015.
- "High Performance Computing: Problem Solving with Parallel and Vector Architectures", G. Sabot. Addison Wesley, 1995.
- Fortran Standards Web site, https://wg5-fortran.org
- "Fortran For Scientists and Engineers", S. Chapman. McGraw-Hill, 2017;
- Fortran 90 guide, http://www.fortran90.org/

References (4)

- "Introduction to Computational Economics using Fortran", H. Fehr and F. Kindermann. OUP, 2018;
- "Parallel Programming with Co-Arrays", R. Numrich. CRC Press, 2018;
- Fortran Wiki, http://fortranwiki.org
- "Modern Fortran", Milan Curcic. Manning Publications, 2019;
- "Scientific Software Design: The Object-Oriented Way", D. Rouson. Cambridge University Press, 2014;

Let's Link Up

Ways to connect with us

Twitter:

www.twitter.com/NAGTalk

Blog:

http://www.nag.co.uk/blog

LinkedIn:

http://www.linkedin.com/e/vgh/2707514/