

Fortran Modernisation Workshop

University of Cambridge, 12-13 April 2018

Fatima Chami (`fatima.chami@stfc.ac.uk`)

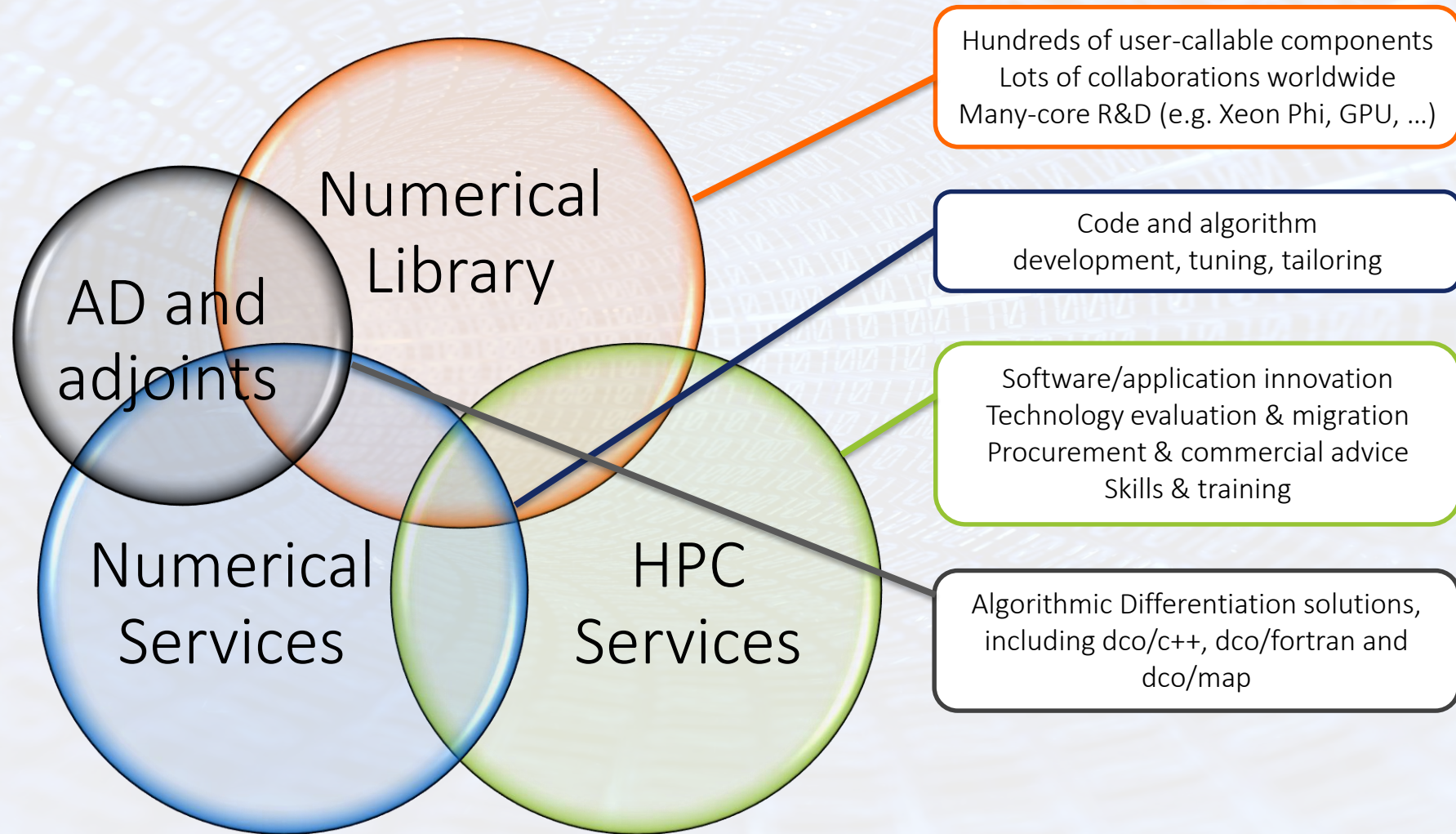
Wadud Miah (`wadud.miah@nag.co.uk`)

Viktor Mosenkis (`viktor.mosenkis@nag.co.uk`)

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 (e.g. AMD, Intel)

NAG Products & Services

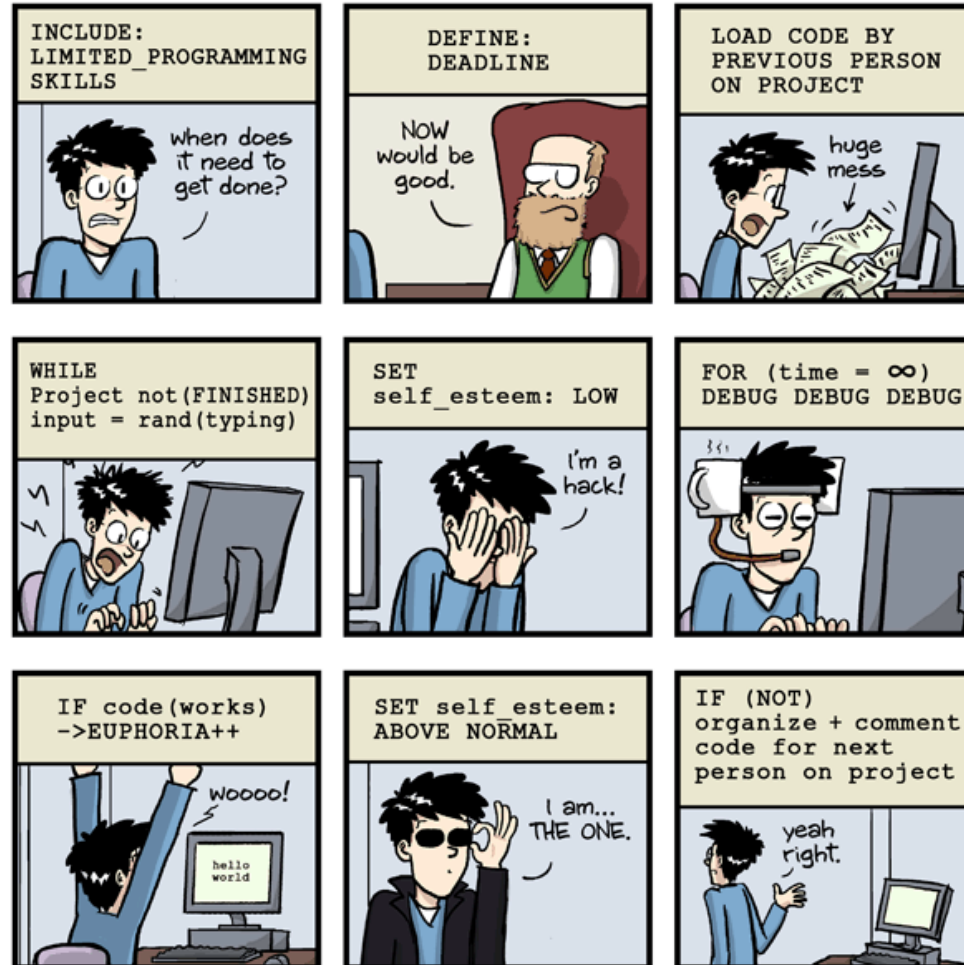


Reusing This Material

- This work is licensed under a Creative Commons Attribution. Non-Commercial-ShareAlike 4.0 International License:
http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US
- This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original;
- Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Programming by Scientists

PROGRAMMING FOR NON-PROGRAMMERS



<http://phdcomics.com>

JORGE CHAM © 2014

WWW.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;
- Using Makefile for building and Fortran Documenter for code documentation;
- 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. High Performance Fortran parallelism use was very limited and later abandoned – 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;
- The 2018 (formerly known as 2015) standard has yet to be formalised;
- This workshop will be discussing Fortran 90, 95, 2003 and 2008 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, Oracle, Cray, Numerical Algorithms Group (NAG), Portland Group (Nvidia), British Computer Society, Fujitsu;
- Academia: New York University, University of Oregon, George Mason 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, IBM, Oracle, Lahey, PathScale and Absoft;
- 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 - see reference [1] for further details. This reference is kept fairly up to date.

[1] http://www.fortranplus.co.uk/resources/fortran_2003_2008_compiler_support.pdf

Source Code Formatting

- Write code that is both *clear to readers and the compiler*;
- You are also writing code for a debugger, profiler, and testing frameworks;
- Easy to read code makes it easier for the compiler to optimise;
- From Fortran 90, free form formatting is provided. This means code can be placed in any column and can be 132 characters long;
- Write code that is as simple as possible and avoid coding “tricks” that obscure algorithms;
- Comment your code well particularly when you are writing complex code for your community;
- Name your subroutines, functions and variables that are *meaningful to your scientific community*.

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 code¹, 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 clear;
- Capitalise names of constants:

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

¹Exceptions apply

Coding Style Suggestions (2)

- Use *two whitespaces 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

Coding Style Suggestions (5)

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

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

- Always use 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;

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 at the beginning of the file with a) purpose of code. Include LaTeX code of equation b) author and email 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 )
```

Naming Conventions (2)

- Avoid generic names like `tmp` or `val` even in functions/subroutines that have a scope outside more than one block;
- Loop 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 $p = \rho RT$:

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

- In functions and subroutines use the `intent` keyword when defining 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
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;
- Use compiler flag to check for out of bounds memory reference¹;

¹Consult your compiler documentation

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(:)
```

- In some occasions, array operations are more likely to vectorise than their loop equivalents.

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
```

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-thresholdn` where *n* is between 0 and 100;

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;
- *To the best of our knowledge, no other compiled language has the Fortran 90 array feature.*

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))

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

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
```

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
```

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 by a subroutine or function:

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

- When an argument needs to be written by a subroutine or function:

```
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;
- Make sure your subroutines are compact enough which makes it easier to debug. Testing a subroutine is known as a *unit test*.

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
```

Source Code Documentation

- Documentation for codes is usually seen as a peripheral activity;
- Instead, it should be seen as intrinsically part of code development;
- A separate document can contain the documentation for the code, but it quickly gets out of date and is difficult to synchronise with the code which is a dynamic entity;
- Solution? Self-documenting code. As well as previous recommendations, use comments to describe the code;
- Keep the documentation up to date as out of date comments can confuse the code developers.

What Should be Documented?

- Every *program, module, submodule, derived data type, function* and *subroutine* should be documented;
- For a program, the documentation should describe what the program does and any references to external documentation, e.g. academic papers, user guides, code web page, book chapter;
- For modules and submodules, the purpose of the module, a brief description of the functions and subroutines it contains, and the variables it uses;

What Should be Documented?

- Use LaTeX syntax if required. Source code documenting systems such as Fortran Documenter can render the equations;
- Any block of code that needs explanation – this is left to the coder.

Documenting Functions and Subroutines

- A description of the function and subroutine, and what equation it solves. Use LaTeX syntax if required;
- A description of all the arguments passed to the function or subroutine. Use the `intent` keyword which gives additional information;
- Describe any algorithms used and any external references;
- A function's purpose is to return a value, so no arguments should be modified - only `intent(in)`;
- If an argument needs to be modified, then one should instead use a subroutine indicating which argument will be modified.

Memory Management (1)

- Fortran 90 introduced dynamic memory management which allows memory to be allocated at run time;
- Always use dynamic memory allocation as 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 leaks¹;
- *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;*

¹Use 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!*

Memory Optimisations

- *Always use unit stride when allocating memory, e.g. **do not use:***

```
real, dimension(1:N:4) :: mesh
```

- *Instead allocate **contiguous** memory:*

```
real, dimension(1:N) :: mesh
```

- 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.

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 `contiguous`:

```
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
```

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 something known as Cray pointers;
- A pointer is an object that points to another variable which is stored in another memory location;
- Always assign it to null, so it is in a known state:

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

- Pointers are sometimes used to avoid expensive copy operations;

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 optimise 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 optimising operations on them.

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

- 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 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, not just Fortran 90;
- Files ending with `.F90` are pre-processed before being compiled. The Fortran pre-processor command is `fpp`;
- 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;
- Pre-processor is mainly used to build on different platforms and takes longer to compile.

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.

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)

- 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.

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 can occur!

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

- The last three are potentially dangerous as serious loss of precision could occur.

Precision Bugs (1)

- The following code segments have bugs:

```
real(kind=REAL32) :: a, geom, v, g_p  
a = geom * v ** (2/3) ! calculate surface area  
g_p = 6.70711E-52
```

```
real(kind=REAL64) :: theta  
real(kind=REAL32) :: x  
x = 100.0_REAL64 * cos( theta ) ! mixing of precisions
```

Precision Bugs (2)

```
real(kind=REAL64) :: d  
real(kind=REAL32) :: x, y  
d = sqrt( x**2 + y**2 )
```

- Compilers are generally not good at spotting precision bugs;
- The FPT [1] tool can detect precision bugs.

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 Modules

```
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 (1)

- When naming internal modules, use the `mod` suffix:

```
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 end the function, subroutine, types, modules with the name as shown above, e.g. `end module Matrix_mod`. This helps delineate the block;
- *Modules allow type checking for function/subroutine arguments at compile time so errors are quickly identified;*

Fortran Module Names (2)

- 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;
- There is no standard for the module file format - every compiler implements them different, e.g. Intel in binary, GNU and NAG in text format.

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**.

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
```

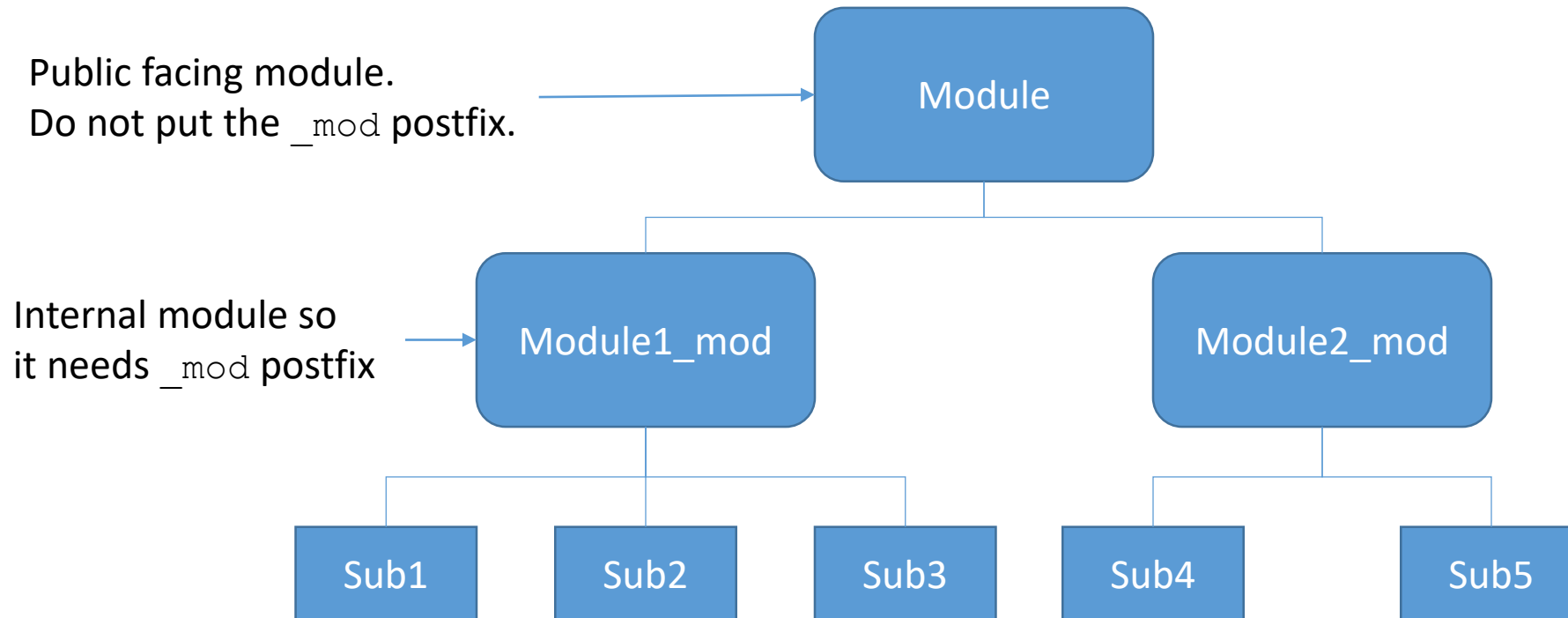
The diagram illustrates the relationship between the module and the program. A blue arrow points from the `vector_mod` module definition to the `use vector_mod` statement in the program. Another blue arrow points from the `real_sum` function definition to the `my_sum(vecr)` call in the program. A third blue arrow points from the `int_sum` function definition to the `my_sum(veci)` call in the program.

```
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 Module Hierarchy

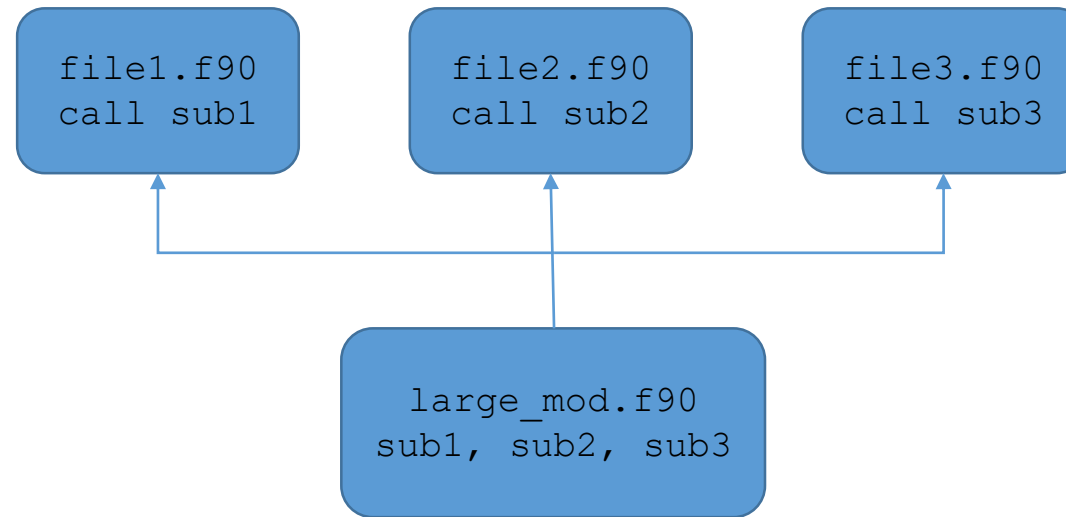


Fortran Submodules (1)

- Fortran 2008 introduced the submodule feature which allows the separation of a) function, subroutine and variable *declarations* (Fortran interfaces) 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;

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
```

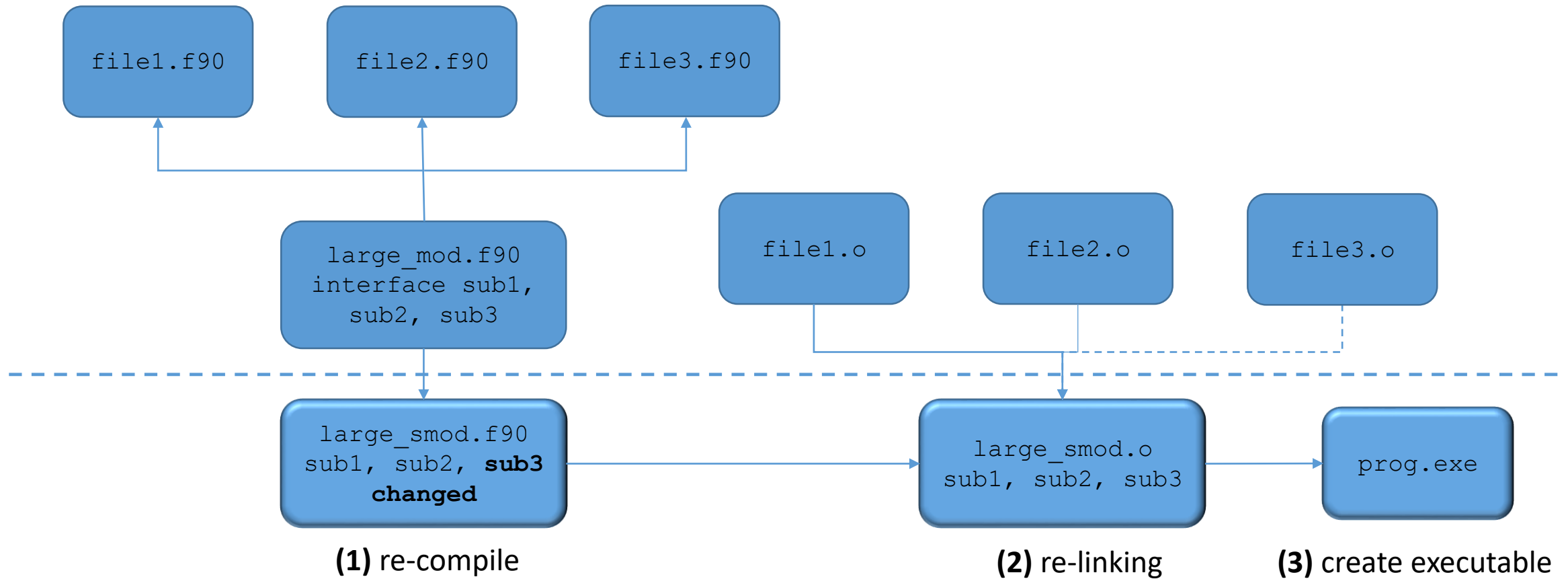
```
    a = a**2
```

```
  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

- 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 optimise (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 optimisations;
- Avoid `while`, `do until` and `repeat until` loops. These loops are sometimes required, e.g. for iterative algorithms that continue until a solution (within error bounds) is achieved.

Forall Loops

- Fortran 95 `forall` loops are like DO loops except that loop iterations are completely independent;
- This allows the compiler to parallelise and/or vectorise:

```
!$omp parallel workshare
forall ( i = 1:100 )
    vec(i) = vec1(i) + vec2(i)
end forall
!$omp end parallel workshare
```

Do Concurrent Loops (1)

- Fortran `forall` has been obsoleted in the 2018 standard due to performance issues (implicit barrier after each statement);
- Like `forall` loops, do concurrent loops can be executed completely independently;
- A new construct has replaced `forall` called `do concurrent` (2008):

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

- The compiler is likely to multi-thread or vectorise the above.

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
```

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 $\sqrt{-1}$, operation involving ∞ , NaN operand	Quiet NaN (not a number)
IEEE_OVERFLOW	Rounded result larger in magnitude than largest representable format	$+\infty$ or $-\infty$
IEEE_UNDERFLOW	Rounded result smaller than smallest representable format	Subnormal or flushed to zero

IEEE Compiler and System Support

- There is no standardised way to handling floating point exceptions in Fortran. Floating point exceptions are 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.`

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 (1)

- 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 compiler to optimise code.

API Design (2)

- 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 )
```

API Design (3)

- Use the `result` clause when defining functions:

```
function delta( a, b ) result ( d )  
    real, intent(in) :: a, b  
    real :: d  
  
    d = abs( a - b )  
end function delta
```

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 optimisation;
- *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 optimisation (`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 to not 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

Deleted

	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	15
Shared DO termination and termination on a statement other than END DO or CONTINUE	90	15

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:**

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
```

try to avoid GO TOs

- 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 (*,*)`

- [Write a message to the appropriate unit and then] read from the appropriate unit

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 or CONTINUE

Obsolescent

	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	15+	-
COMMON and EQUIVALENCE statements and BLOCK DATA program unit	15+	-
Specific names for intrinsic functions	15+	-
FORALL construct and statement	15+	-

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
```

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
```

- Use integer return with IF or SELECT CASE construct

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 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 and keywords. X must be {C, L, U};
- `-margin=N` - Set the left margin (initial indent) to N (usually 0);

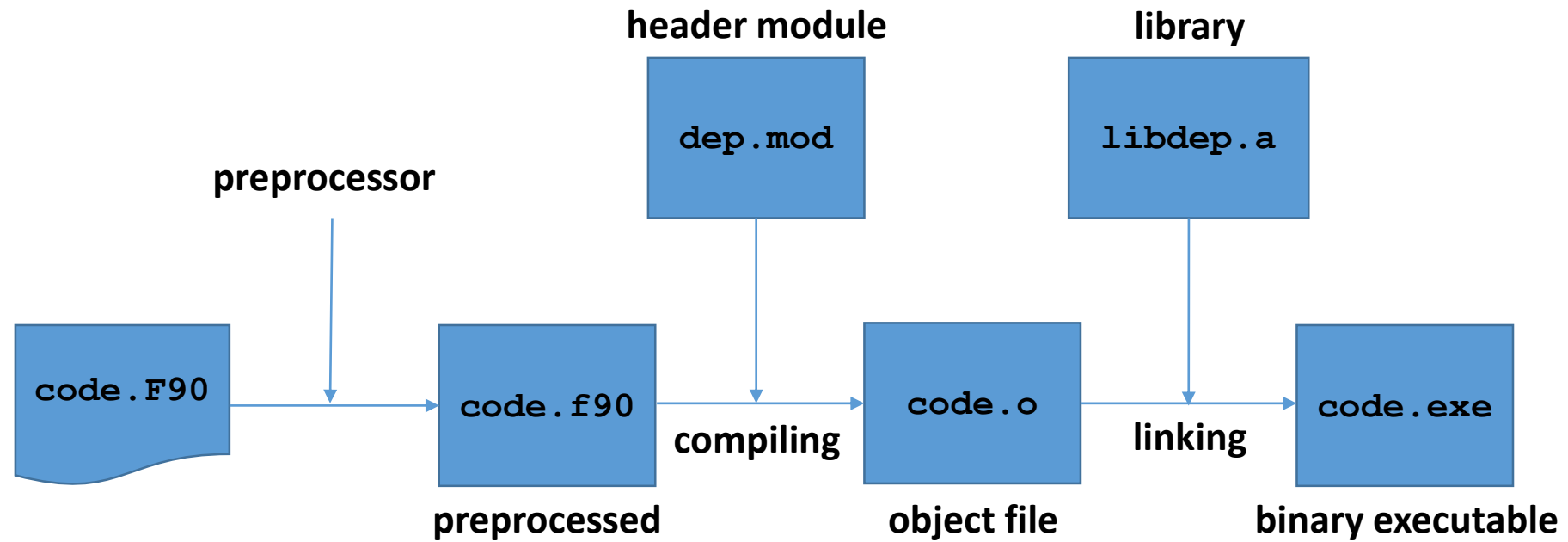
NAG Fortran Compiler Polish (2)

- `-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}`;

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`

Building of Codes



Build Commands (1)

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

```
nagfor -I/path/to/mod -c 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/lib -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;

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;
- To determine which shared libraries are required:

```
[wadudm@nash ~/fortran]$ ldd workshare.exe
```

```
linux-vdso.so.1 => (0x00007ffc6ebdf000)
```

```
libgfortran.so.3 => /lib64/libgfortran.so.3  
(0x00002b046d5d2000)
```

```
libm.so.6 => /lib64/libm.so.6 (0x00002b046d8fa000)
```

Build Commands (3)

- Statically linking reduces the time the executable code gets loaded into memory. Subsequently, the shared 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;*
- 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 -lnetcdf -lnetcdf \
-o code.exe
```

- The `netcdf` 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 (`fmw` 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 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;
- 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;
- Release 6.2 has just been released and 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
```

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
```

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 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
```

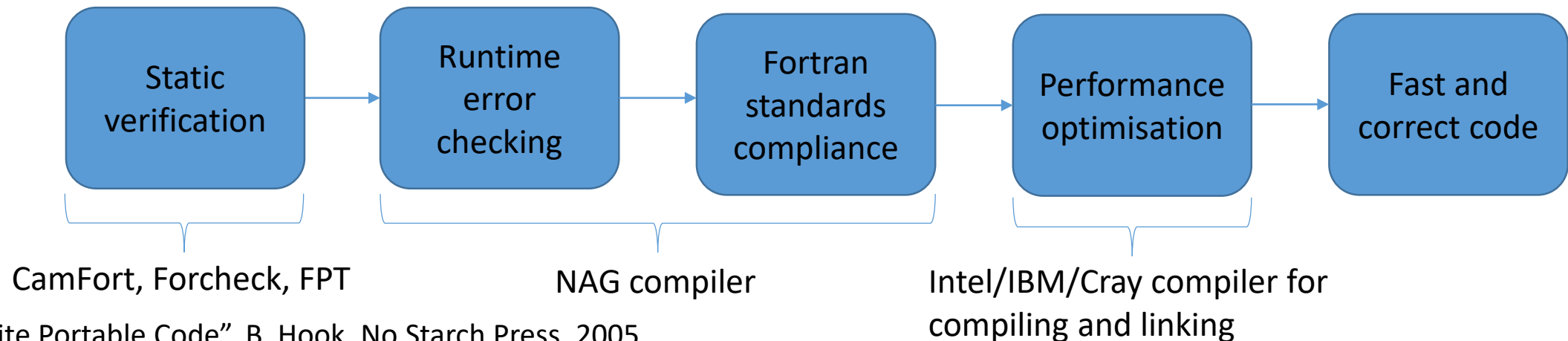
- Runtime output:

```
Runtime Error: ovf_test.f90, line 7: INTEGER(int32)  
overflow for 12312312 * 12312312
```

```
Program terminated by fatal error
```

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 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. It simply checks the Linux file time stamp;
- 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.

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 optimisation flags.

Compiling a Fortran Module

- 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 parameter *symbols*. The path to header module file is specified in the `-I` flag during compilation, e.g. `-I/home/miahw/dep/include`
- `Mesh_mod.o` which is an object file which contains all functions and subroutines as *symbols* for linking with main code;
- 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`

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
    ifort $^ $(LDFLAGS) -o $@                  # (3)

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

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

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:

```
make -n
```

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 n processes, use the command:

```
make -j  $n$ 
```

- This must be done in the same directory with the file called `Makefile`.

Make Dependencies - `makedepf90`

- The Fortran `use` keyword is used to determine the dependencies between source code files;
- These dependencies are then used to create the rules in a Makefile;
- This can be tedious for a large number of source code files;
- The `makedepf90` [1] utility can calculate the dependencies automatically;
- Simply invoke the command in the source code directory:

```
makedepf90 *.f90
```

[1] <http://personal.inet.fi/private/erikedelmann/makedepf90/>

makedepf90 - Output

```
[miahw@bengal solutions]$ makedepf90 *.f90
```

```
CFL_mod.o : CFL_mod.f90 Types_mod.o
```

```
fd1d_heat_explicit.o : fd1d_heat_explicit.f90 \  
Solver_mod.o IO_mod.o CFL_mod.o RHS_mod.o Types_mod.o
```

```
IO_mod.o : IO_mod.f90 Types_mod.o
```

```
RHS_mod.o : RHS_mod.f90 Types_mod.o
```

```
Solver_mod.o : Solver_mod.f90 RHS_mod.o Types_mod.o
```

```
Types_mod.o : Types_mod.f90
```

Code Documentation

- Code documentation is important and part of the code. The documentation will increase the code's impact and longevity;
- Code documentation – *the code itself with comments*. Fortran Documenter allows developers to quickly navigate around the code;
- User guide – a guide on how to use the code for new users;
- Installation guide – how users should build and install the code on their desktop and HPC clusters. List any dependencies on external libraries, e.g. BLAS, LAPACK.

Fortran Documenter (FORD)

- There are a few documentation tools available for Fortran (e.g. Doxygen, f90doc, ROBODoc);
- Doxygen has been designed for C, C++ and other languages, and has been “hacked” to document Fortran and lacks key features;
- FORD [1] has been designed exclusively for modern Fortran;
- Module dependencies are visualised and Fortran keywords are coloured;
- Main configuration file is required and uses Markdown syntax. This can include environment variables in the form `${USER}`;

[1] <https://github.com/cmacmackin/ford>

FORD Key Features

- Extracts information about variables, procedures (functions and subroutines), procedure arguments, derived data types, programs and modules;
- Source code can be preprocessed;
- Supports LaTeX equations using MathJax;
- Searchable documentation using Tipue Search which supports a wide range of Web browsers. Can search source code as well as procedures, variables, etc;
- Can add GitHub (Bitbucket), Twitter and LinkedIn pages;

FORD Key Features

- Source code can be downloaded;
- Links to other parts of the source code, e.g. subroutines and derived data types;
- Symbols (e.g. main code, modules, derived data types) are appropriately coloured;
- Developed in Python and installed using the PIP Python package manager, e.g. `pip install ford`
- Output is in HTML so can be viewed by a Web browser;

FORD Configuration (**fmw .md**)

src_dir: ./src (source directory - always use relative paths)
output_dir: ./doc (documentation directory - **use different paths**)
project_github: <https://github.com/cmacmackin/ford>
project: Fortran Modernisation Workshop
summary: Solves 1D heat equation
author: Jon Doe
author_description: Senior computational physicist
email: jon.doe@email.com

FORD Configuration

`docmark: !` (marker for recognising comments **after** statements)

`predocmark: >` (marker for recognising comments **before** statements)

`graph: true` (displays call trees, module dependencies)

`source: true` (displays source code on documentation page)

`search: true` (adds search feature)

`version: 1.2.3` (version of your code)

`display: public` (what to display - must be on separate lines)

`protected`

`private`

FORD Configuration

- After key-value pairs are assigned, code notes and bugs can be listed;
- To write a note, use:

```
@Note
```

```
This version of the code writes in ASCII. Need to write  
in NetCDF
```

- To write list of bugs in code:

```
@Bug
```

```
Need to fix I/O issue (#143)
```

- Any Markdown content can be placed here.

FORD Code Documentation

- Documentation can contain links and LaTeX equations in text:

`y` is calculated using `\(y = x^2 \)`

`x` is calculated using `\[x = sqrt{y/z} \]`

Use `$$ p = \rho RT $$` for the equation of state

- Or even labelled equations on separate lines:

```
\begin{equation}
```

```
PV = nRT
```

```
\end{equation}
```

- Save the configuration in `fmw.md` and then invoke the command `ford fmw.md` and the HTML file will be in `doc/index.html`

Documenting Code

- Comments can be placed **after** the code using `!!`:

```
real, dimension(:) :: x
```

```
!! solution vector
```

- Or can be placed **before** the code using `!>`:

```
!> solution vector
```

```
real, dimension(:) :: x
```

- Although both styles are acceptable, comments **before** the code are recommended. The `!>` marker is also used by Doxygen;
- More importantly, document the code whichever style you use!

Documenting Code

- To document variables, including subroutine arguments:

!> initialized to the value of π

```
real(REAL64) :: global_pi
```

- To document derived data types:

!> particle data type to contain position

```
type particle
```

```
    real(REAL64) :: x, y, z
```

```
end type particle
```

Documenting Code

- To document subroutines - must use backslash and parenthesis for LaTeX equations

!> This subroutine solves $c = \sqrt{a^2 + b^2}$

```
subroutine square( a, b, c )
```

!> a is the length

```
real, intent(in) :: a
```

!> b is the height

```
real, intent(in) :: b
```

!> c is the solution

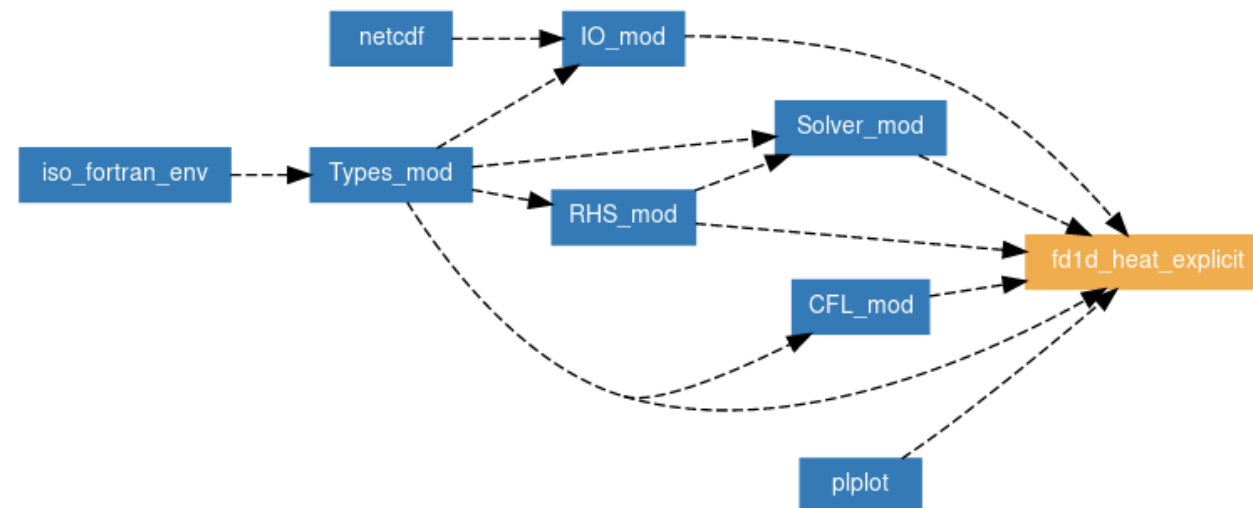
```
real, intent(out) :: c
```

```
end subroutine square
```


FORD Output (1)

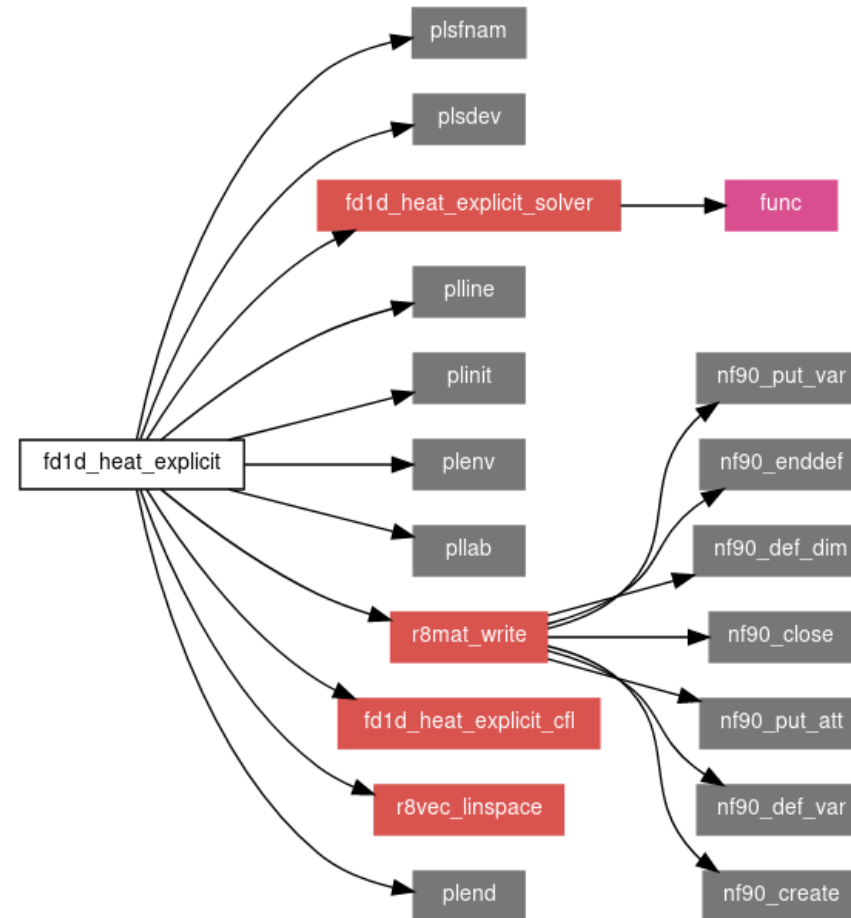
Modules

Module	Source File	Description
CFL_mod	CFL_mod.f90	this module calculates the CFL number
IO_mod	IO_mod.f90	this module deals with netcdf I/O
RHS_mod	RHS_mod.f90	this module contains the right-hand side of the PDE
Solver_mod	Solver_mod.f90	this module solver the PDE $\frac{\partial \mathbf{H}}{\partial t} - K \frac{\partial^2 \mathbf{H}}{\partial x^2} = f(x)$
Types_mod	Types_mod.f90	this module contains the real and integer kind variables



FORD Output (2)

- Stack trace



Fortran in LaTeX Documents (1)

```
\documentclass{article}
\usepackage{xcolor}
\usepackage{listings} % uses listings package
\lstset{language=[08]Fortran, % Fortran 2008 standard
basicstyle=\ttfamily,
keywordstyle=\color{blue}, % key words are blue
commentstyle=\color{green}, % comments are green
captionpos=b, % caption is at the bottom
numbers=left, % line numbering
numberstyle=\small\color{black} % size and colour
}
```

Fortran in LaTeX Documents (2)

```
\begin{document}
```

Listing~\ref{code:f90} shows the Fortran code.

```
\begin{lstlisting}[caption={Test code},label={code:f90}]
```

```
program latex_test
```

```
  implicit none
```

```
    integer :: i ! loop counter
```


```
    real :: vector(1:100)
```

```
end program latex_test
```

```
\end{lstlisting}
```

```
\end{document}
```

Listing 1 shows the Fortran code.



```
1 program latex_test
2   implicit none
3
4   integer :: i ! loop counter
5   real :: vector(1:100)
6 end program latex_test
```

Listing 1: Test code

End of Day 1 - Start Exercises

- Exercise code is at `fmw_exercises/src/fd1d_heat_explicit.f90`
- Copy over `fmw_exercises/exercises.pdf` to your laptop/desktop for instructions
- Presentation can be found at - also copy over to your laptop/desktop `fmw_exercises/FortranModernisationWorkshop.pdf`

Day Two Agenda

- Serial NetCDF and HDF5;
- Using pFUnit for unit testing;
- Git version control and PLplot visualisation;
- Introduction to parallelisation in MPI, OpenMP, Global Arrays and CoArrays. GPU programming using CUDA Fortran and OpenACC;
- Introduction to the NAG numerical library;
- Fortran interoperability with R, Python and C;
- Fortran verification with CamFort.

Data Management



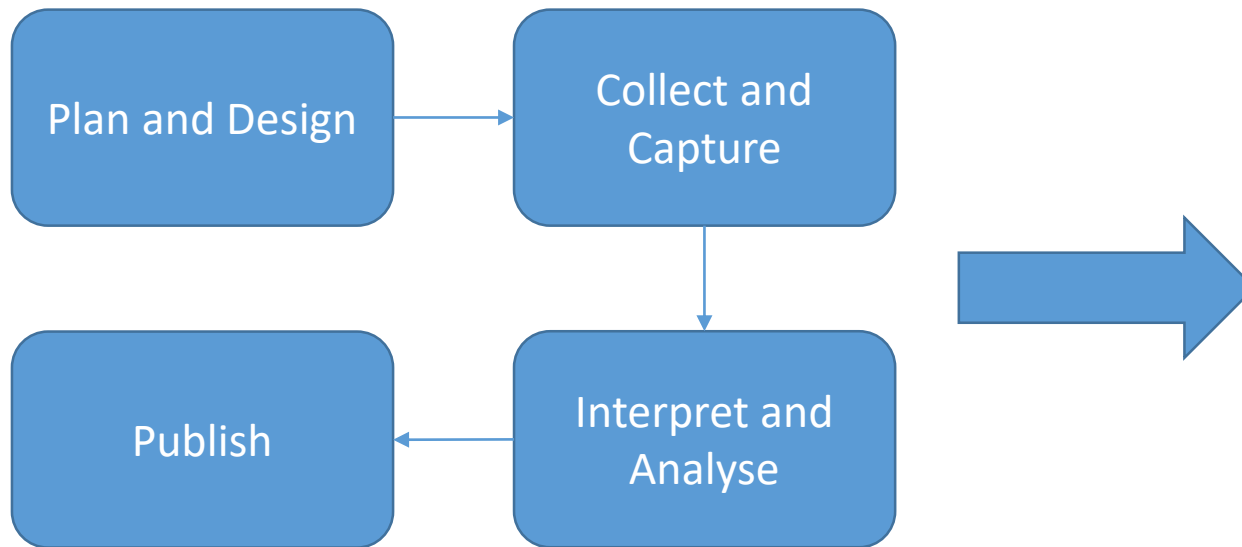
<http://phdcomics.com>

Data From Simulations

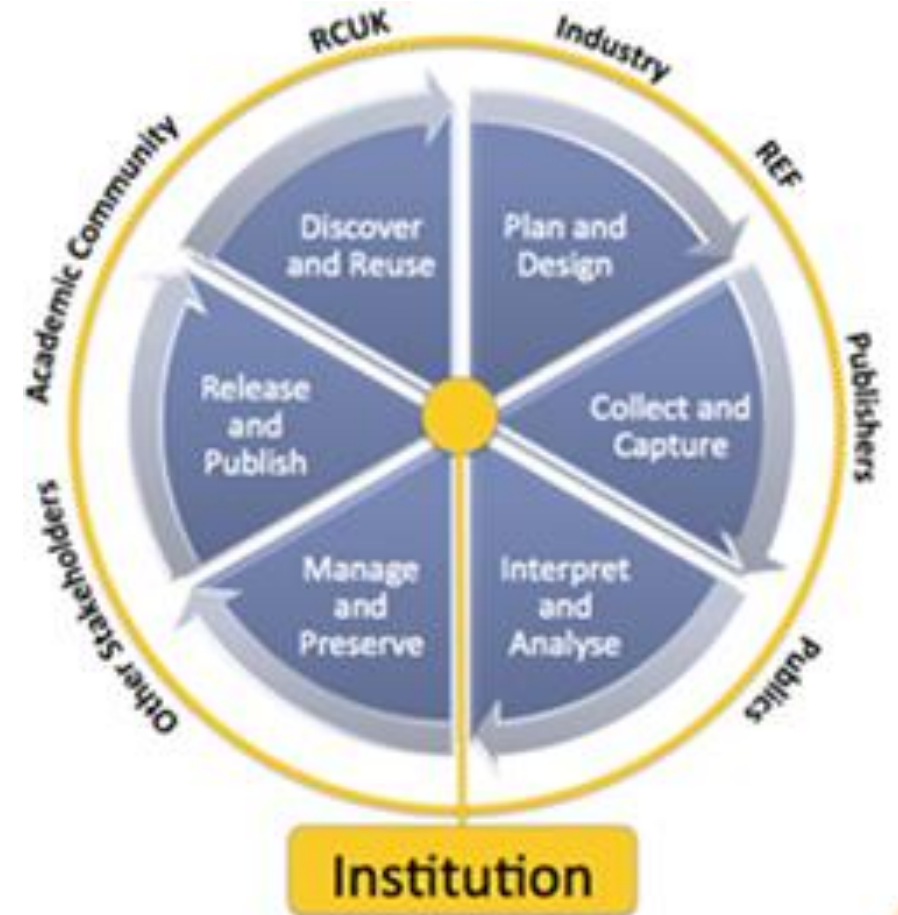
- Computational codes are producing petabytes of data from multiple simulations creating a large number of data sets;
- Data is stored for two reasons: checkpoint/restart for fault resiliency and, visualisation and analysis. *If used for visualisation, consider using single precision as this will halve 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

Old Model



New Model



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

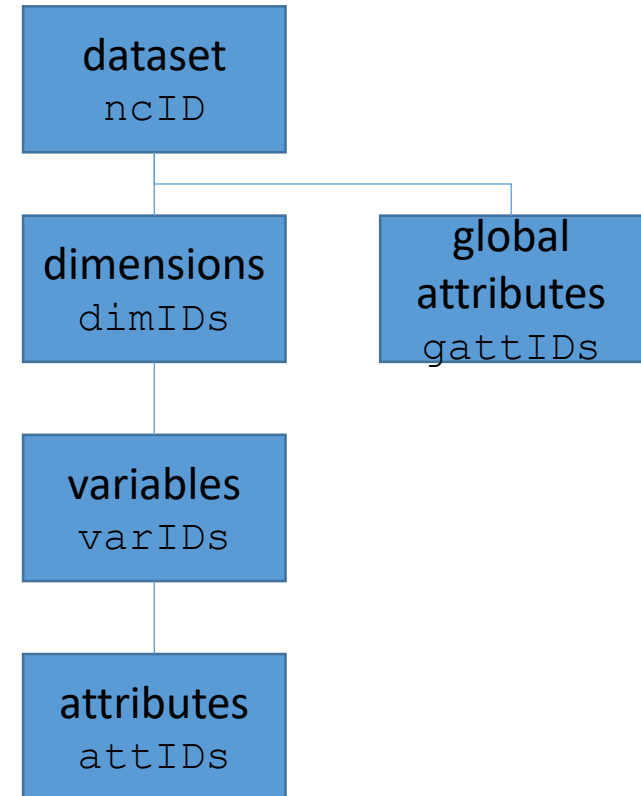
- Stores data in the form of multi-dimensional arrays;
- Underlying storage is abstracted away from user applications;
- Is portable across many different architectures, hence allows collaboration. It can be read by codes in other programming languages;
- Uses a highly optimised indexing system so data access is direct rather than sequential;
- Applies compression techniques to minimise file sizes;
- 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/m²;
- 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_NO_CLOBBER`. 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;
- The open function `NF90_OPEN`, has similar arguments as the create function;
- To close a data set, simply invoke:

```
function NF90_CLOSE( ncid )
```

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;

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
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)

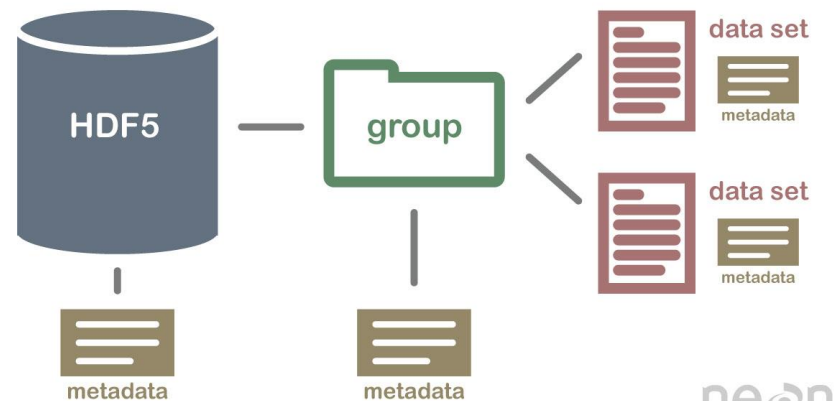
- `ncrename` - renames dimensions, variables and attributes in a NetCDF file;
- `ncra` - averages record variables in arbitrary number of input files;
- `ncwa` - averages variables in a single file over an arbitrary set of dimensions with options to specify scaling factors, masks and normalisations;
- `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 than NetCDF.

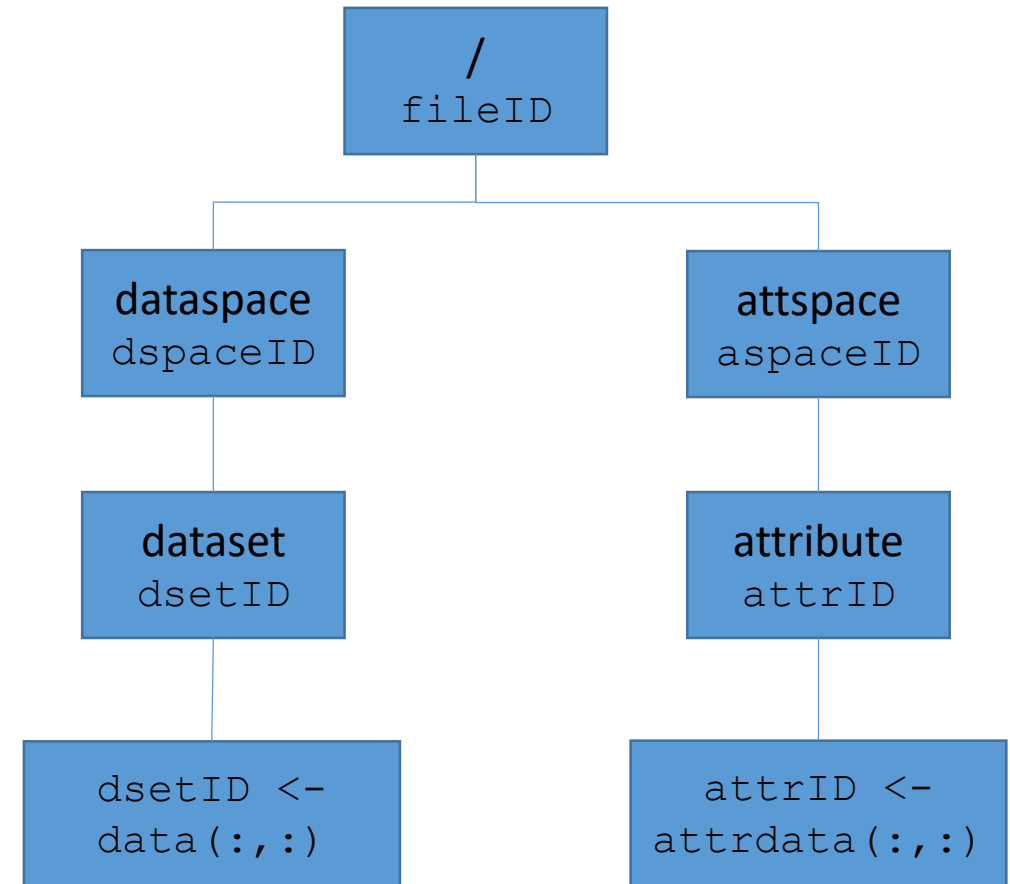
HDF5 Data Model

- **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 )
```


Testing Code

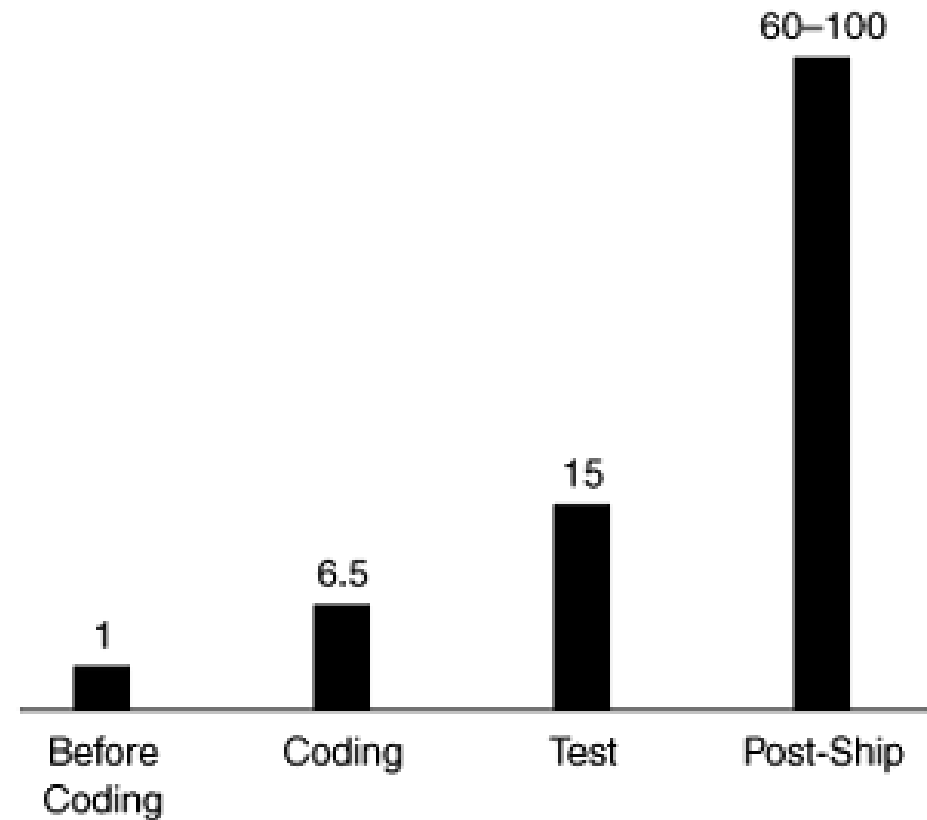


Testing Code

- For complex code, at least one unit test per possible path in a function or subroutine is advised to ensure high-levels of code coverage;
- Integration tests which verify the entire code;
- Regression tests are required to ensure a fix or other modification of a code has not inadvertently broken another part of the code;
- The greater the number of regression tests and the higher the code coverage, the more likely regression tests are able to detect when a fix has broken existing functionality.

Cost of Defects in Codes

- The greatest cost of code defects is when it is detected by academics and causes the retraction of academic papers [1];
- There have been high profile cases of retracted papers due to code defects so find defects as early as possible [2]:



[1] “A Scientist’s Nightmare: Software Problem Leads to Five Retractions”, G. Miller, Science, vol 314, no 5807, 2006.

[2] “Software Engineering, A Practitioner’s Approach”, R. Pressman. McGraw Hill, 1992.

Testing Scientific Codes (1)

- Testing scientific codes is difficult due to the inherent uncertainties/errors contained in the solution;
- Errors in scientific models ϵ_{model} , e.g. Navier-Stokes;
- Errors caused by domain discretisation ϵ_{disc} ;
- Truncation errors in numerical algorithms $\epsilon_{\text{algorithm}}$, e.g. Taylor series. Number of iterations in iterative algorithms;
- Implementation errors of the numerical algorithm, e.g. software bugs;
- Numerical rounding and truncation errors ϵ_{float} , from floating point data types. This is also affected by parallelism and vectorisation;
- Errors propagate in time marching schemes and need to be bounded;

“Accuracy and Reliability in Scientific Computing”, B. Einarsson. SIAM, 2005.

“Verification and Validation in Scientific Computing”, W. Oberkampf and C. Roy, Cambridge University Press, 2010.

Testing Scientific Codes (2)

- Numerical value cannot be exactly compared with exact value but is bounded by:

$$||\mathbf{u}_{\text{numerical}} - \mathbf{u}_{\text{exact}}||_{\infty} < \epsilon_{\text{float}} + \epsilon_{\text{algorithm}} + \epsilon_{\text{model}} + \epsilon_{\text{disc}} = \epsilon$$

- In unit testing, a heuristic approach is taken when selecting ϵ and obtaining an accurate value is very difficult;
- If ϵ is too large, faults will go undetected. If it is too small, it will create false positives;
- This falls within the area of *uncertainty quantification* which is a new area of research.

Testing Scientific Codes (3)

- When testing numerical code, *never test for equality between floating point numbers whatever the precision*;
- Instead do `abs (a - b) < tol` which is an accepted level of error tolerance;
- This is due to rounding errors in digital computers:
 $(a + b) + c \neq a + (b + c)$ and $(a * b) * c \neq a * (b * c)$
- $RD(x)$ - round towards $-\infty$
- $RU(x)$ - round towards $+\infty$
- $RZ(x)$ - round towards zero
- $RN(x)$ - round to nearest representable number in radix 2. *This is the default* which can be changed in Fortran 2003.

Testing in Computational Science (1)

- Unit tests should test individual subroutines and functions. These should be executed at every commit or merge request - *unit testing*;
- Test each component with other components that it interacts with - *integration testing*;
- Solution verification 1 - does the solution satisfy the differential equation which has an analytical solution? This will take longer and should be executed less frequently - *acceptance testing*;
- Solution verification 2 - does the solver converge with known initial conditions? This will also take longer and should be executed less frequently - *acceptance testing*;

Testing in Computational Science (2)

- Mesh convergence testing - refine mesh and apply verification 1 and 2 above;
- The infinity norm should be calculated and be within a certain tolerance;
- When testing multi-dimensional PDEs, reduce the simulation by one dimension for testing, e.g. run a one-dimensional solution for a two dimensional PDE at a spatial slice;
- *Model validation cannot be automated.* It must be visualised and interpreted.

Testing Tools for Fortran (1)

- The compiler – prints a lot of diagnostic information as well as status of compilation;
- NAG Fortran compiler does extensive runtime testing, including Fortran standards compliance tests;
- FORCHECK [1] – performs full static analysis and standards conformance. Diagnostics is more comprehensive than compilers;
- FPT [2] – mismatched arguments, loss of precision. Code metrics;
- `gcov` – checks the coverage of your unit tests;
- Valgrind or RougeWave MemoryScape for memory leaks;

[1] <http://www.forcheck.nl>

[2] <http://www.simconglobal.com>; [3] <http://www.cl.cam.ac.uk/~dao29/camfort/>

Testing Tools for Fortran (2)

- Eclipse Photran plugin does static analyses;
- CamFort for dimensional analysis on variables [3];
- pFUnit – unit testing framework for serial and parallel (OpenMP, CoArray and MPI) codes.

Unit Testing

- "A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behaviour of that unit of work" [1];
- For our purposes *a unit is a single Fortran subroutine or function*;
- The unit test is narrow, specific and tests disparate parts of code;
- Tests are independent and do not cause side effects. Order of tests does not determine results and uses limited resources;
- Ideally, all functionality is covered by at least one test. This is known as test coverage.

[1] <http://artofunittesting.com/>

Testing Frameworks (1)

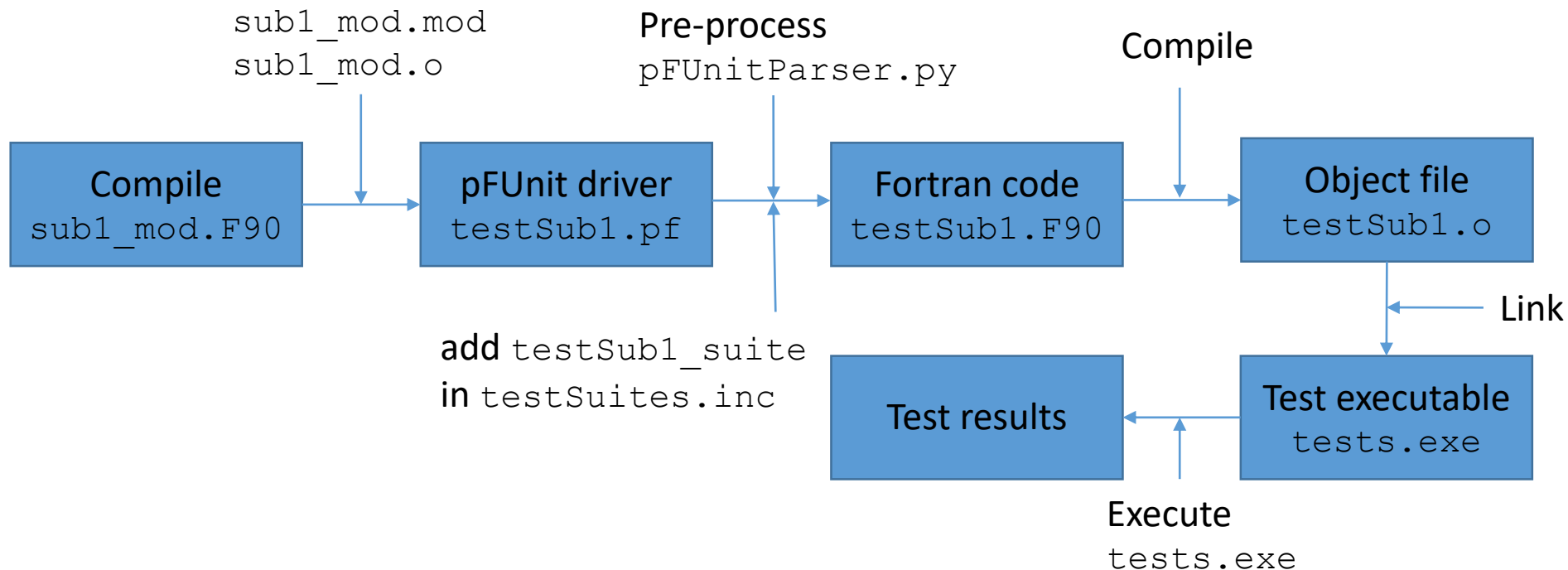
- The code that tests your subroutines and functions is known as a *test driver code*;
- This can be normal Fortran code but it is recommended to use a testing framework such as pFUnit;
- Testing frameworks *standardise* the testing process and how driver codes are written and increases portability of your test driver code;
- Frameworks have a standard way to print results (pass/fail) and also print the execution time of the test;
- A number of tests can be bundled into a single executable, thus simplifying the testing process;

Testing Frameworks (2)

- For parallel codes, MPI initialisation and finalisation is done automatically;
- It is recommended to test all subroutines and functions in a module and bundle them into a single executable

pFUnit 3.0 - Unit Testing in Fortran

- Test driver codes implemented in *pseudo* Fortran;
- Tests sequential and parallel codes, e.g. OpenMP and MPI;
- Enables parameterised tests with extensible OOP Fortran;



Test Driver Code

- The test driver is a pseudo Fortran of code that tests your functions and subroutines. Below code is stored as `testCode.pf`

```
@test
subroutine testCode( )
  use pFUnit_mod      ! required
  use test_mod        ! contains Riemann subroutine

  real :: result2, tol = 0.00001
  call Riemann( 2.0, 1.0, result2 )

  @assertEqual( result2, 3.5, tol ) ! [1]
end subroutine testCode

[1] abs( result2 - 3.5 ) <= tol
```

Process for Testing

- In configuration file `testSuites.inc` add line:

```
ADD_TEST_SUITE( testCode_suite )
```

- Pre-process driver code:

```
$PFUNIT/bin/pFUnitParser.py testCode.pf testCode.F90 -I.
```

- Compile the created Fortran code [1]:

```
$FC -O0 -I$PFUNIT/mod -c testCode.F90
```

- Create the `tests.exe` executable binary:

```
$FC -o tests.exe -I. -I$PFUNIT/mod $PFUNIT/include/driver.F90 \  
testCode.o code_mod.o -L$PFUNIT/lib -lpfunit -O0
```

- Execute binary executable `./tests.exe` which will print result of tests.

[1] `$FC` is the Fortran compiler with all optimisations switched off

pFUnit Output

- The output of pFUnit is very similar to the output of other unit testing frameworks for other programming languages:

```
CFL stability criterion value =    0.320000
```

```
Time:                0.001 seconds
```

```
OK
```

```
(1 test)
```

- The time and result of the test is printed.

Code Coverage

- Ideally, tests should cover 100% of code;
- To measure amount of code coverage in tests, use `gcov` tool for the `gfortran` compiler;
- Replace `$FC` in previous example with `gfortran -fprofile-arcs \ -ftest-coverage` which is required for compilation and linking;
- After executing binary `tests.exe`, execute `gcov test_mod.F90` which will *print percentage of code covered by test [1]*;
- It also creates a text file `test_mod.F90.gcov` which annotates the code with which lines have been executed and how many times.

[1] `.gcno` and `.gcda` files are also created

Fortran Syntax Checkers for Linux Editors

- Fortran syntax checkers also exist for traditional Linux editors such as vim and Emacs which check 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>

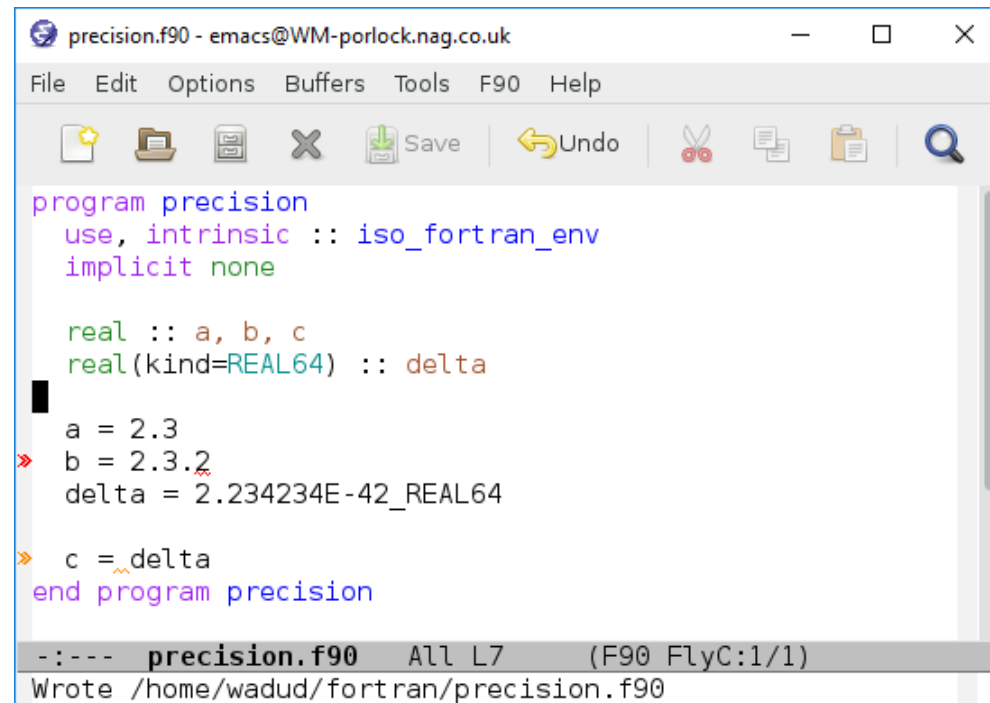
Flycheck for Fortran (1)

- 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)



The screenshot shows an Emacs editor window titled "precision.f90 - emacs@WM-porlock.nag.co.uk". The menu bar includes File, Edit, Options, Buffers, Tools, F90, and Help. The toolbar contains icons for new file, open file, save, close, save all, undo, redo, cut, copy, paste, and search. The code in the buffer is as follows:

```
program precision
  use, intrinsic :: iso_fortran_env
  implicit none

  real :: a, b, c
  real(kind=REAL64) :: delta

  a = 2.3
  » b = 2.3.2
  delta = 2.234234E-42_REAL64

  » c = delta
end program precision
```

At the bottom of the window, a status bar displays: "-:--- precision.f90 All L7 (F90 FlyC:1/1)" and "Wrote /home/wadud/fortran/precision.f90".

- 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

Code Directory Structure (1)

- `src/` - where the source code resides;
- `ext/` - location for external libraries;
- `tests/` - test driver code;
- `doc/` - where documentation is placed;
- `bin/` - where the binary executables are placed;
- `lib/` - where shared and static libraries are placed;
- `include/` - where header modules are placed;

Code Directory Structure (2)

- `Makefile` - The file that will build your code;
- `README` - description of code, contacts, web address;
- `INSTALL` - installation instructions;
- `ChangeLog` - revision history of code;
- `LICENCE` - type of licence. Legally, no one can use your code unless you specify a licence.

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 -lp1plotfortran -lp1plot \  
-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.

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>

Fortran Command Line Arguments Parser (2)

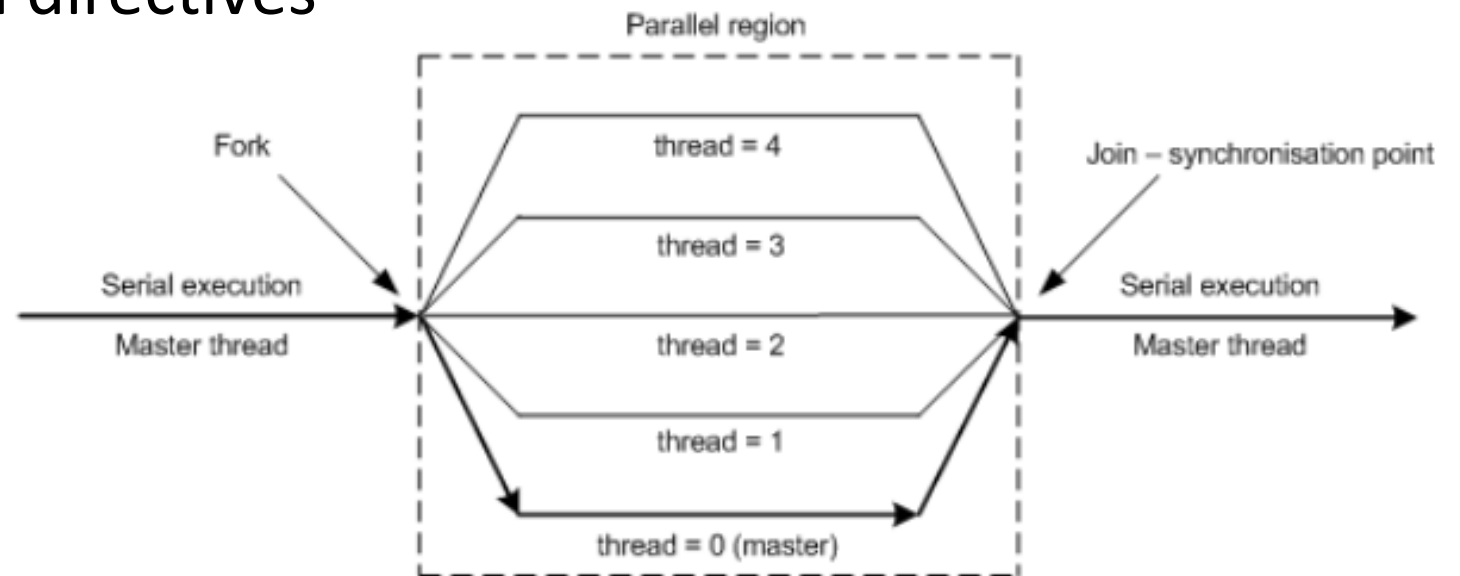
```
call cli%init(description = 'minimal FLAP example')
call cli%add( switch = '--int', switch_ab = '-i', &
              help = 'an integer (number of intervals)', &
              required = .true., act = 'store', error = ierr )
call cli%add( switch = '--tol', switch_ab = '-t', &
              help = '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

- 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



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
```

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
```


Shared memory (3)

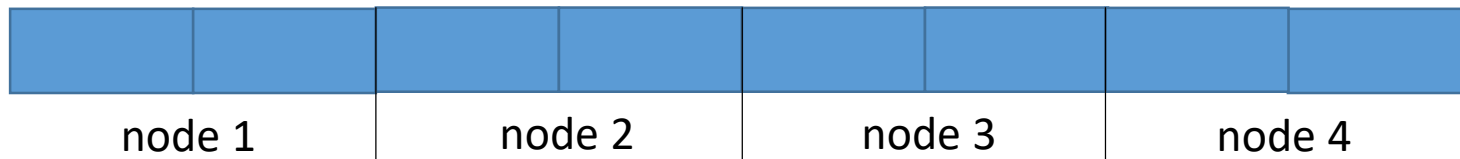
```
01 !$omp parallel default(none), shared(n, a, X, Y, Z), private(i)
02 !$omp do
03     do i = 1, n
04         Z(i) = a * X(i) + Y(i)
05     end do
06 !$omp end do
07 !$omp end parallel
```

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
```

Distributed memory

- Single Program Multiple Data (SPMD);
- Parallel *across nodes* and/or cores within node;
- PGAS (Partitioned Global Address Space) [1]:
 - CoArray Fortran and GA (Global Arrays) - similar to Unified Parallel C (UPC)



- MPI (Message Passing Interface) [2]



[1] www.pgas.org

[2] www.mpi-forum.org

CoArrays (1)

- *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 (2)

```
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
```

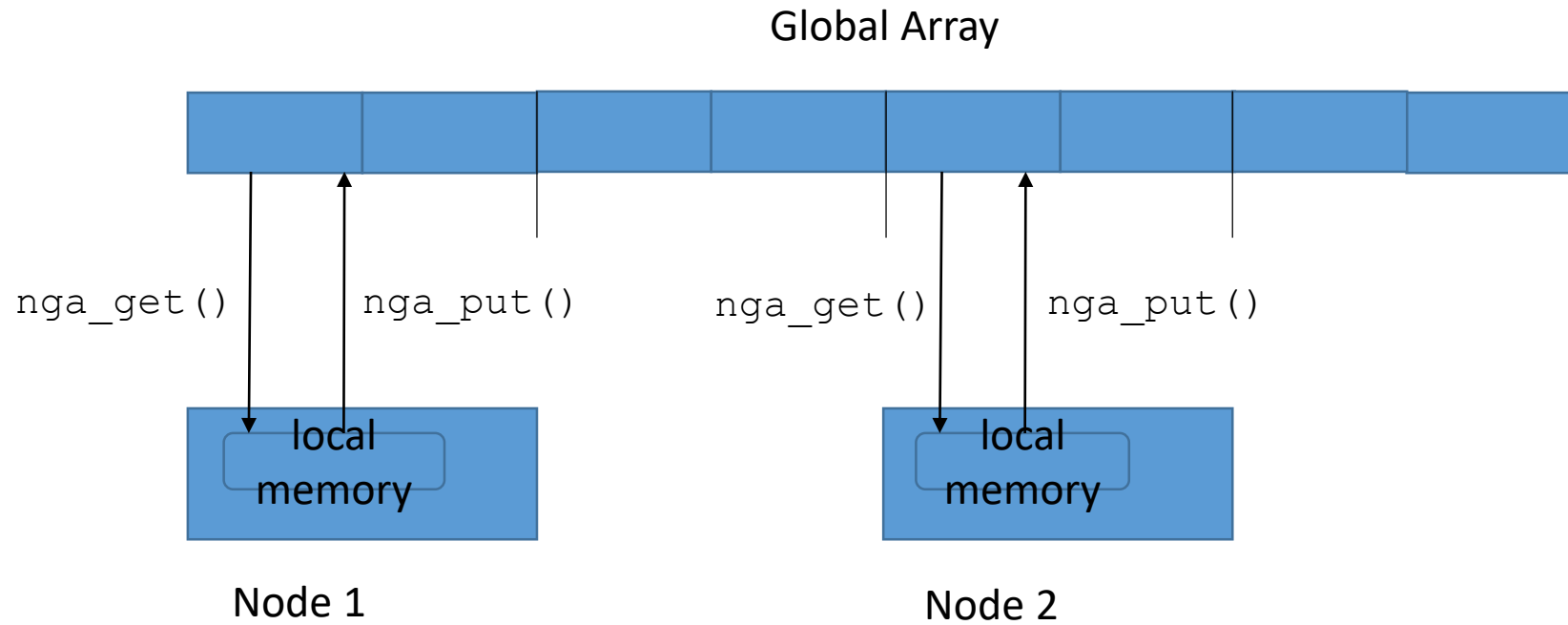
CoArray 2018

- Collective operations are implemented as subroutines:
`co_broadcast`, `co_max`, `co_min`, `co_sum` and `co_reduce`
- Fault tolerance has also been implemented using the `failed_images` intrinsic function which returns an array of failed images;
- If an image has failed, then it is up to the developer on how to deal with this failure;
- Can configure teams of images, much like communicators in MPI.

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.

Global Arrays (2)



Global Arrays (3)

```
01  call mpi_init(err)
02  call ga_initialize()
03  nProcs = ga_nNodes()
04  procN = ga_nodeId()
05  print *, "I am process", procN, "of", nProcs
06  call ga_terminate()
07  call mpi_finilize()
```

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

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



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)
```

Fortran 2008 MPI Bindings

- MPI Fortran bindings have been re-written that use the Fortran 2008 bindings;
- MPI data types are proper parameterised data types much like in the C MPI bindings;
- One should use the Fortran 2008 bindings for any new MPI codes:

```
use mpi_f08
```

- For existing codes, continue to use the Fortran 90 bindings:

```
use mpi
```

- At all costs, avoid the Fortran 77 bindings:

```
include 'mpif.h'
```

Fortran 2008 MPI Interface

```
interface MPI_Send
  subroutine MPI_Send_f08ts(buf , count , datatype , &
    dest , tag , comm , ierror)
    use :: mpi_f08_types , only : MPI_Datatype , MPI_Comm
    implicit none
    type(*) , dimension (..), intent(in) :: buf
    integer , intent(in) :: count , dest , tag
    type(MPI_Datatype), intent(in) :: datatype
    type(MPI_Comm), intent(in) :: comm
    integer , optional , intent(out) :: ierror
  end subroutine MPI_Send_f08ts
end interface MPI_Send
```

MPI 3.1 Implementation Status

	MPICH	MVAPICH	Open MPI	Cray MPI	Tianhe MPI	Intel MPI	IBM BG/Q MPI ¹	IBM PE MPICH ²	IBM Platform	SGI MPI	Fujitsu MPI	MS MPI	MPC	NEC MPI
NBC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	(*)	✓	✓
Nbrhood collectives	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
RMA	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	Q2'17	✓
Shared memory	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	*	✓
Tools Interface	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	*	Q4'16	✓
Comm-creat group	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	*	✓
F08 Bindings	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✗	Q2'16	✓
New Datatypes	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
Large Counts	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	Q2'16	✓
Matched Probe	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	Q2'16	✓
NBC I/O	✓	Q3'16	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	Q4'16	✓

OpenACC (1)

- Only the PGI, CAPS 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

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 optimisation;
- 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];

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)

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

CUDA Fortran for DO Loops (2)

- Reduction is automatically generated:

```
rsum = 0.0  
!$cuf kernel do <<<*, *>>>  
do i = 1, nx  
    rsum = rsum + a_d(i)  
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.

Vectorization (2)

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

01 do i = 1, n

02 $Z(i) = a * X(i) + Y(i)$

03 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 (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
```

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 - assumed shaped arrays are not currently supported** - proposed in Fortran 2018;

Fortran Kind Type	Equivalent C Type
<code>C_INT</code>	<code>int</code>
<code>C_FLOAT</code>	<code>float</code>
<code>C_DOUBLE</code>	<code>double</code>

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
000000000000000000 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_f
      real(kind=C_FLOAT) :: x(*)
      integer(kind=C_INT), value :: n
    end function sum_c
  end interface
  integer, 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 `reciprocal_c` which is listed in the symbol table:

```
nm sum_c.o
```

```
00000000000000000000 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
```

Memory Layout in C and Fortran

- Remember that in Fortran, arrays are column-major, i.e. `vec(i, j)`, `vec(i+1, j)`, `vec(i+2, j)` are contiguous;
- Whereas in C, they are row-major, i.e. `vec[i, j]`, `vec[i, j+1]`, `vec[i, j+2]` are contiguous;
- When passing multi-dimensional arrays between C and Fortran, they can be transposed for performance;
- However, the transpose operation itself is very expensive. You can use the `transpose` intrinsic function only for 2D arrays;
- *Try to pass one-dimensional arrays to avoid the expensive transpose operation.*

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>

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):

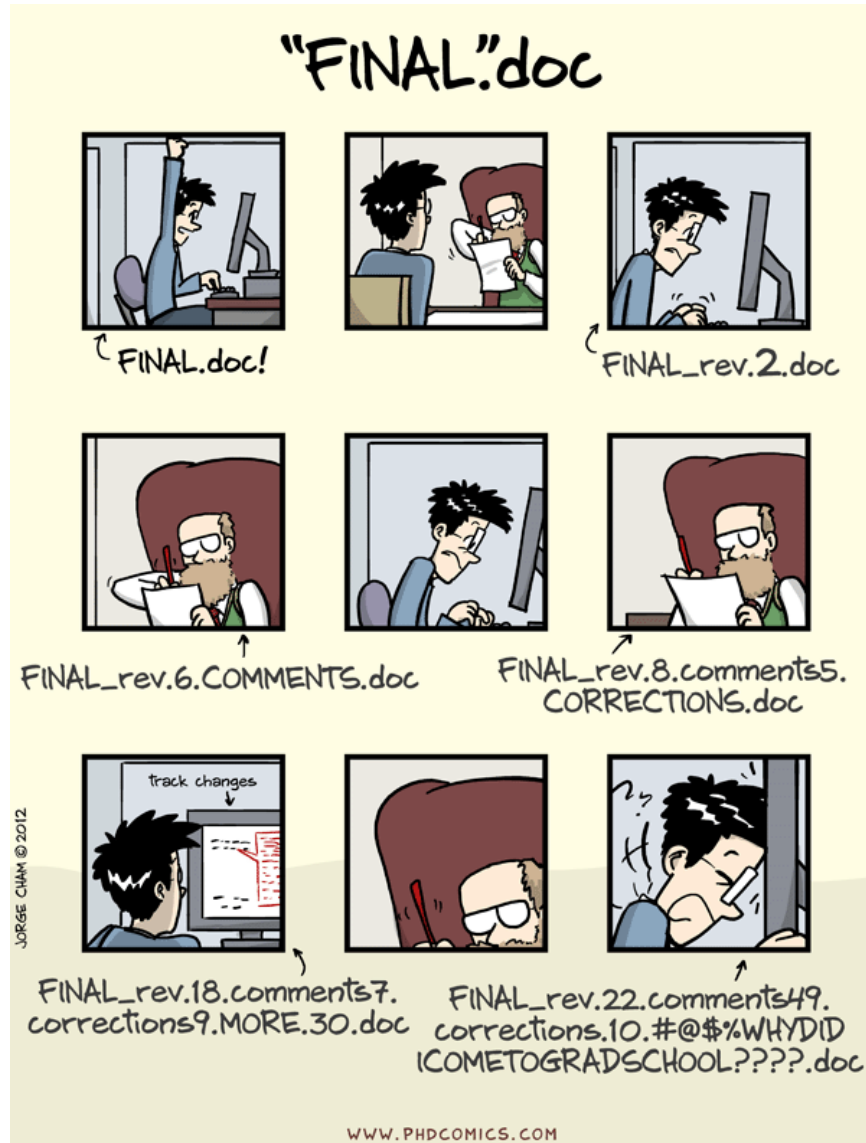
```
gfortran -c sums_mod.F90
gfortran -shared sums_mod.o -o sums_mod.so
```

- 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
```

Version Control



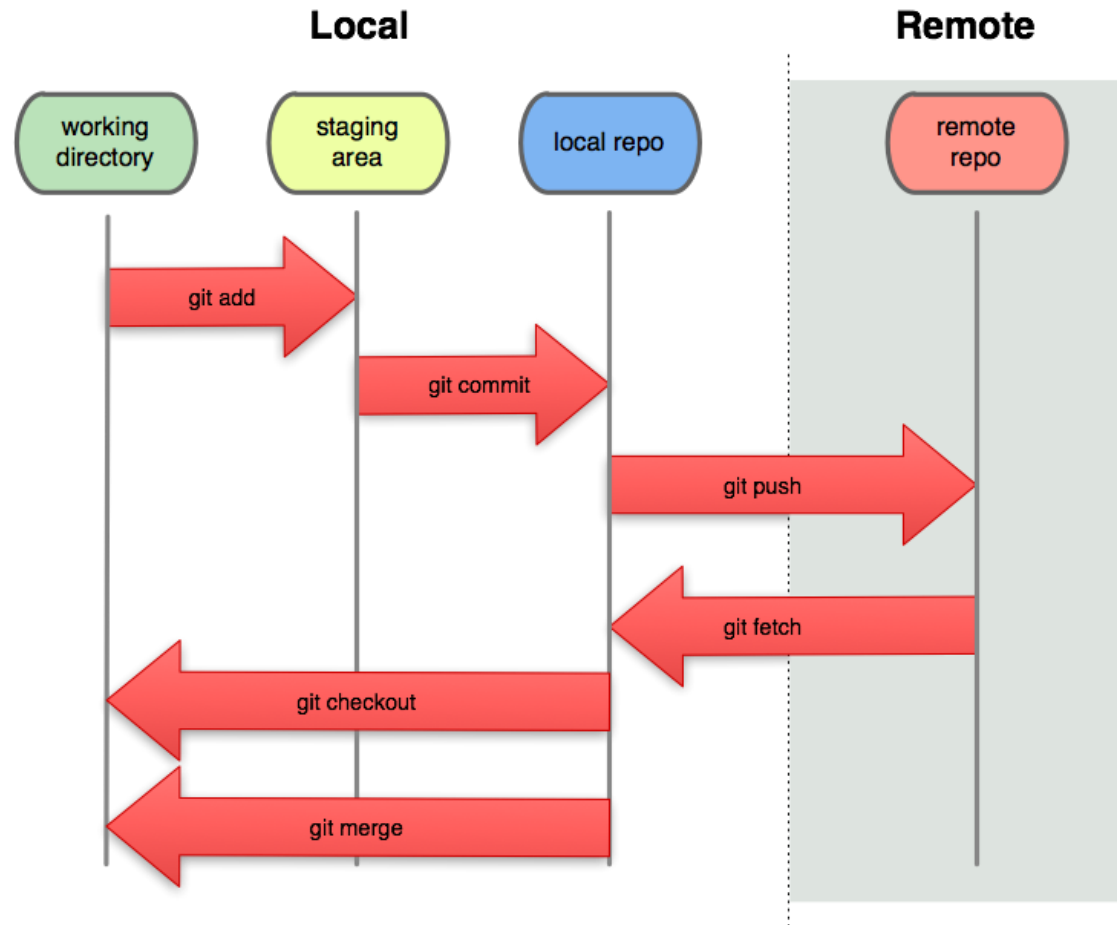
<http://phdcomics.com>

Version Control

- A version control system stores your files and records changes that are made over time and stores meta-data that describe the changes;
- It allows you to load specific versions of your files and monitor changes that are made by a number of developers;
- Anything that is text based and manually created should be version controlled, e.g. source code, Makefiles, documents;
- Anything that can be re-produced should not be version controlled, e.g. datasets, binary executables, libraries;
- The data store is known as a *repository*.

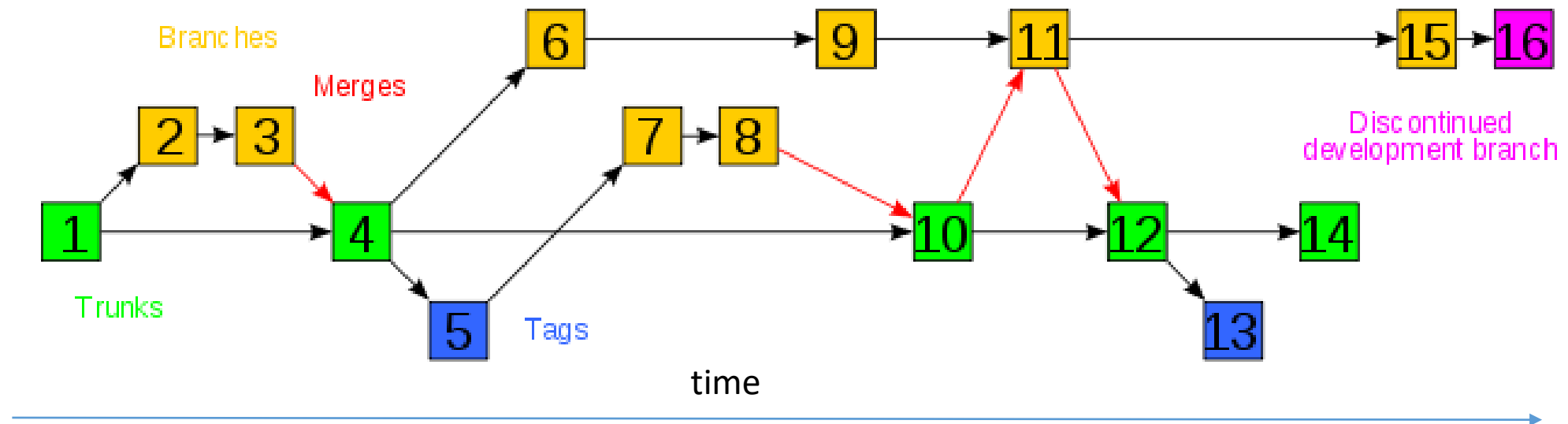
Git Version Control

- Git is a distributed version control system;



Trunk, Branches and Tags

- The *trunk* is the main line of development;
- A *branch* is a duplication of a development tree that allows parallel code development. This can then be merged back from where it branched off;
- A *tag* is a snapshot of development tree at a certain time.



Getting Started With Git

- Set your name and email:

```
$ git config --global user.name "Wadud Miah"
```

```
$ git config --global user.email "wadud.miah@gmail.com"
```

```
$ git config --global color.ui "auto"
```

- Git has extensive help:

```
$ git help <command>
```

where `<command>` is one of many Git operations.

Initialising Git

- Assuming the previous directory structure, in the *root* directory:

```
$ git init .
```

- This will create a `.git` subdirectory that will store Git related files;
- A `.gitignore` file in the root directory contains files (or patterns of files) that should not be tracked - anything that can be re-produced should not be version controlled;
- It should contain an entry on a separate line;
- Comments begin with a hash sign (`#`);

Git Ignore File

- For Fortran, have the following entries in `.gitignore` assuming the directory structure mentioned:

*.o

*.mod

*.a

*.so

*.exe

doc/

include/

bin/

lib/

Git Basics

- To track a file, *add* it to Git:

```
$ git add code.F90
```

- This *stages* your file so you can *commit* it to the *local repository*;

```
$ git commit -m "initial version of main code"
```

```
[master bda6f9a] initial version of main code
```

```
1 files changed, 5 insertions(+)
```

```
create mode 100644 code.F90
```

- *Always* add a commit message using the `-m` flag. You can use multiple `-m` flags for multiple messages;
- Git uses SHA1 hashes as commit numbers.

Git Tracking

- When changing a file, Git will show this:

```
$ git status
```

```
modified:    code.F90
```

- To stage the change, simply use:

```
$ git add code.F90
```

```
$ git commit -m "added a new print statement"
```

```
[master f026b63] added a new print statement
```

```
1 file changed, 1 insertion(+)
```

- Use `git log` to view revision history.

Git Branch

- Every git repository has a *master* branch. To create a branch:

```
$ git branch RB_1.0 master
```

```
$ git branch  
    RB_1.0
```

*** master**

```
$ git checkout RB_1.0
```

```
Switched to branch 'RB_1.0'
```

```
$ git branch
```

*** RB_1.0**

```
    master
```

Git Branch Merge

- When changes are made to the branch, you may want to *merge* the changes back into the *master* branch;

```
$ git checkout master
```

```
$ git merge RB_1.0
```

```
Updating 9a23464..217a88e
```

```
Fast forward
```

```
code.F90 | 15 ++++++
```

```
1 files changed, 15 insertions(+), 0 deletions(-)
```

```
create mode 100644 code.F90
```

Git Tagging

- Git allows tagging which is a method to snapshot a development line;
- Snapshots can be used to tag a code release;
- Use annotated tags that keep metadata such as tagger details:

```
$ git tag -a version-1.4.8 -m "my version 1.4.8"
```

- Use the *major.minor.patch* versioning system [1];
- Then to view the tag:

```
$ git checkout version-1.4.8
```

- To return to the master branch:

```
$ git checkout master
```

[1] <http://semver.org/>

Remote Repository - Bitbucket

- To collaborate with other developers, local repository need to be *pushed* to a remote repository;
- To get other developers' updates, changes need to be *pulled* from remote repository to local repository;

```
$ git remote add origin git@bitbucket.org:user/repo.git
```

```
$ git push origin master # push your changes
```

```
$ git pull origin master # get changes from others
```

- Before making any changes to your local repository, always pull first;
- Push your changes after making your changes to local repository.

Discussion for 5 minutes

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

End of Day Two - Exercises 2

References (1)

- “The Art of Readable Code”, D. Boswell and T. Foucher. O’Reilly, 2011;
- “Modern Fortran in Practice”, A. Markus. Cambridge University Press, 2012;
- “Modern Fortran”, N. Clerman and W. Spector. Cambridge University Press, 2012;
- “Modern Fortran Explained”, M. Metcalf, J. Reid and M. Cohen. Oxford University Press, 2011;
- “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)

- “Read Me First! A Style Guide for the Computer Industry”, Sun Technical Publications. Prentice Hall, 2009.
- “Programming Models for Parallel Computing”, P. Balaji. MIT Press, 2015.