Fortran Modernisation Workshop

Oxford University 13 - 14 July 2016

Kyle Fernandes (kj333@cam.ac.uk)

Fatima Chami (fatima.chami@durham.ac.uk)

Filippo Spiga (fs395@cam.ac.uk)

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

Reusing This Material

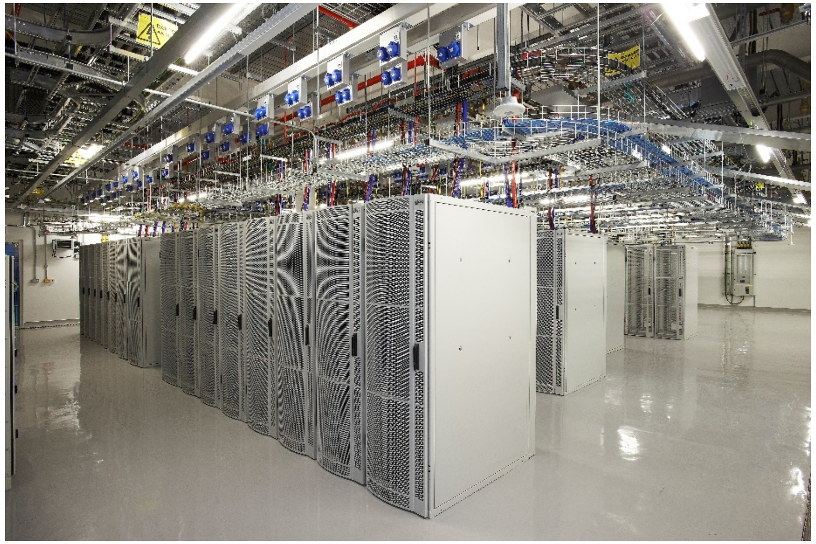
Research Computing Services @ University of Cambridge

UNIVERSITY OF CAMBRIDGE

hpcs

Cambridge HPC service

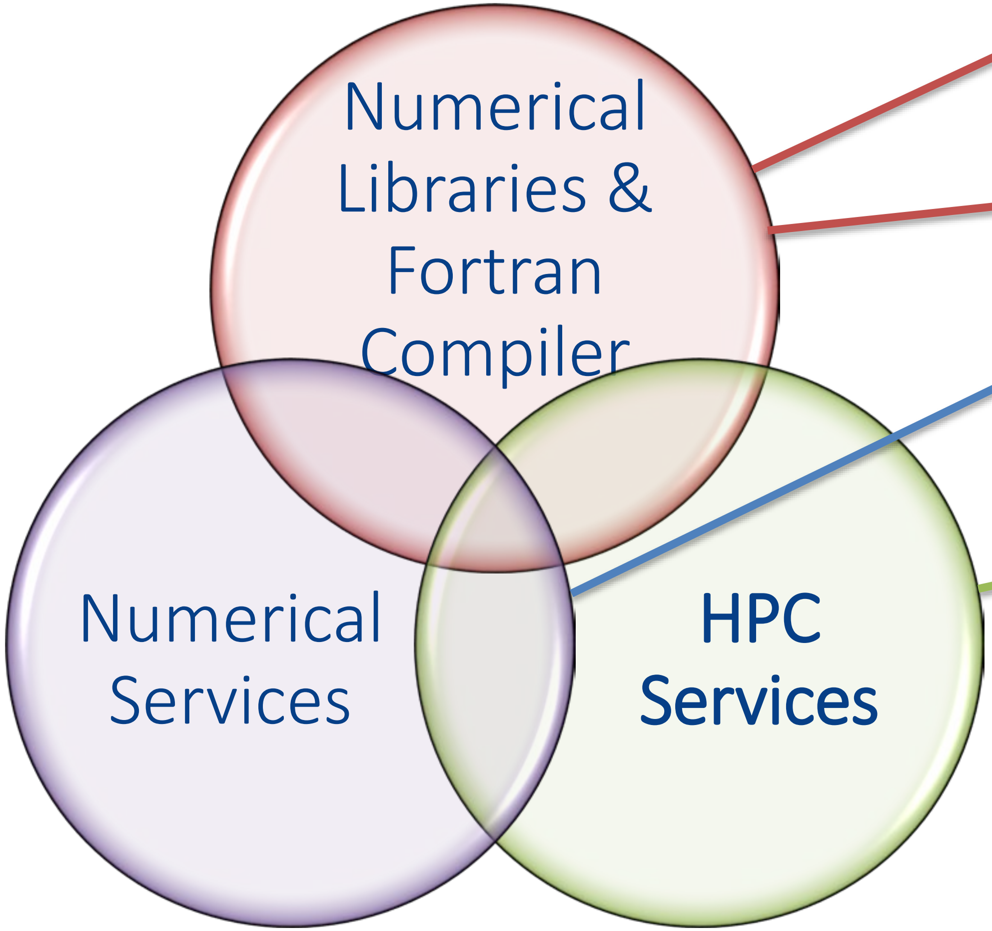
- Around 20 staff
- Research Software Engineering team of 6 people
- Brand-new Data Centre with evaporative cooling
  - **Darwin**: 600 node (9600 cores), dual 8-core 2.6 GHz Sandy Bridge, Mellanox FDR, **~185 TF**
  - **Wilkes**: 128 node (1536 cores), dual 6-core 2.6 GHz Ivy Bridge, 256 card NVIDIA K20 GPU, dual-rail Mellanox Connect IB, **~250 TF** - *Largest GPU cluster in UK academia*
- National Tier-2 Facility, involved in DiRAC, SKA, PSi (industry) projects
- DELL/Intel Solution Centre



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



Over 1700 user-callable components
Lots of collaborations worldwide
Many-core R&D (e.g. Xeon Phi, GPU, …)

Considered as best checking Fortran Compiler

Code and algorithm development, tuning, tailoring

Software/application innovation
Technology evaluation & migration
Procurement & commercial advice
Skills & training

The NAG Library

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

Why use NAG Libraries and Toolboxes?

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

Traditional Uses of The NAG Library

- NAG is used where non-trivial mathematics must be done quickly and accurately on computers
- Largest user groups (not necessarily in order)
  - Academic researchers (typically statistics, applied mathematics, finance, economics, physics, engineering)
  - Engineers (fluid dynamics, large-scale PDE problems, simulations)
  - Quantitative analysts (asset modelling and risk analysis)
  - Statisticians (data mining, model fitting, analysis of residuals, time series, …)

NAG Consulting and Services

ISO 9001 REGISTERED FIRM

- Delivering Expertise and Experience in
  - Numerical Algorithms
  - Numerical Software Engineering
  - High Performance Computing (HPC)
- To customers through
  - Training and mentoring
  - Consulting and advisory services
  - Development or performance projects
  - User / Development / Application support services
  - HPC Technology Intelligence Service

Summary

- Experts in Numerical Computation and High Performance Computing
  - High quality Numerical Software Components
    - Backed up by world-class support
  - Expert numerical services
  - HPC services
    - For managers, users and programmers for the whole life cycle

Results Matter, Trust NAG.

Programming by Scientists

http://phdcomics.com

Day One Agenda

- Software engineering for computational science;
- 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 Doxygen for code documentation;
- Day one practical;
- Supplementary material at www.nag.co.uk/market/training/fortran-modernisation-workshop

Research Software Lifecycle

- Many research codes start from PhD or academic projects;
- They either a) stay within the team (*small*) b) progress to become community code (*medium*) c) used within the discipline (*large*);
- Level of adherence to software engineering (SE) practices will vary according to the size of the code base and discipline;
- One of the aims of SE is to streamline from a small code base to a large code base by ensuring small codes adhere to good SE practices;
- SE is imperative for reproducible and collaborative research;
- *Software engineering practices are applicable for all sizes of research codes!*

Software Engineering Practice (1)

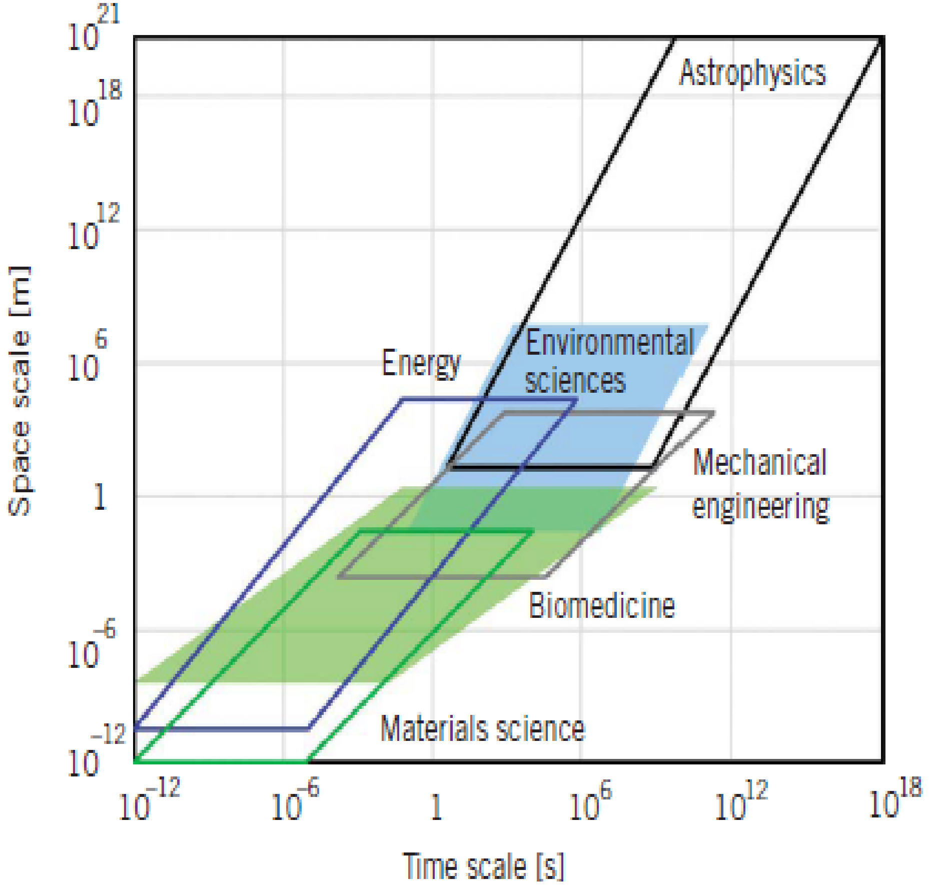
- Goal of research is to find new insights and discoveries in science and engineering;
- Software is an important tool used in research;
- Software engineering (SE) is widely used in computer science and is a mature practice;
- Typical SE has numerous processes and procedures many of which are not relevant for scientific/engineering codes;
- Strictly adhering to the full SE processes can quickly over-burden the academic researcher;
- As an academic, the idea is to be *pragmatic* about software engineering – *use SE practices that are useful to your discipline*;

Why Software Engineering? (1)

- Funding bodies (e.g. EPSRC) now dictate that the code adhere to SE practices for *reproducible* research;
- Aids *collaboration* between different groups;
- Increase chances of obtaining further funding for your research as well as funding for extending your code, e.g. parallelising or adding new features/solvers;
- Will help you develop code so you can spend more time on your science instead of code development headaches;
- For posterity and altruistic reasons by sharing code with the wider scientific community;

Why Software Engineering? (2)

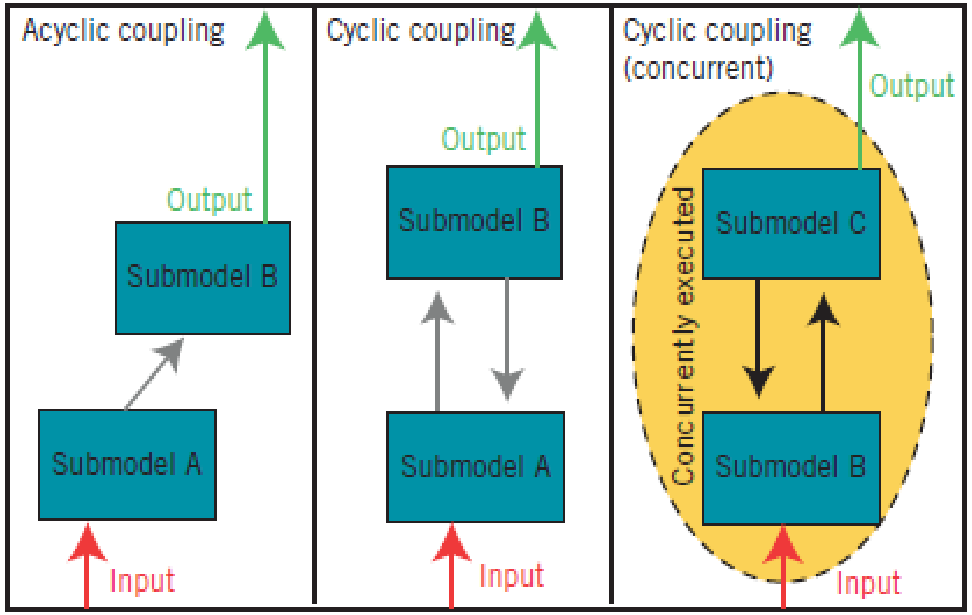
- Computing hardware power and memory has increased dramatically. From MFLOP/s to tens of PFLOP/s, and now heading towards EFLOP/s;
- This has allowed more complex science simulations to be conducted at finer spatial and temporal scales, namely *multi-scale* simulation. For example, from electrons (ps), atoms (100 ps), molecules (100 ns) to continuum modelling (600 seconds, with Δx, Δy, Δz = 50 km);



D. Groen, *et al*, “Survey of Multiscale and Multiphysics Applications and Communities”. *Computing in Science and Engineering*, Vol. 16, No. 2. March/April 2014. pp. 34 - 43

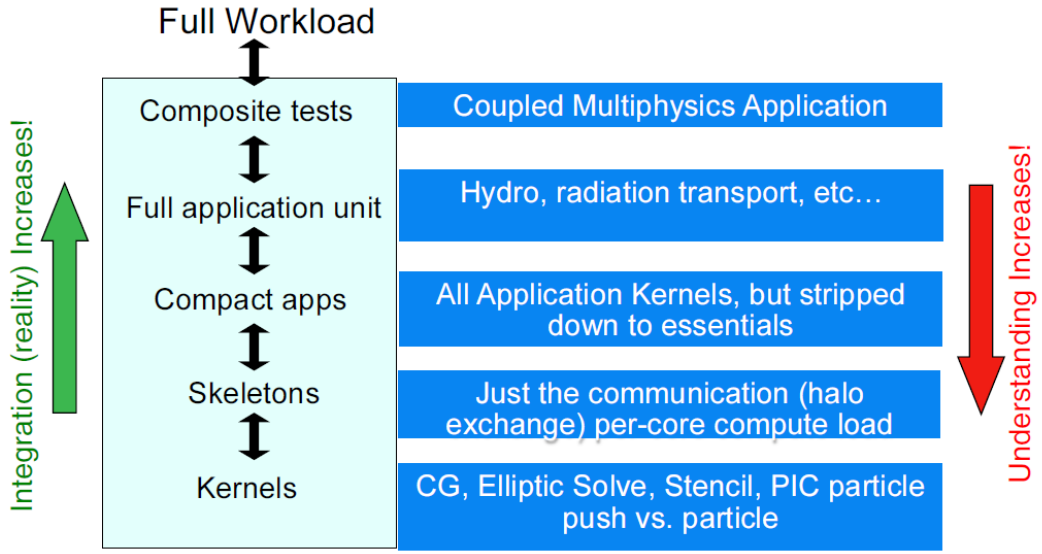
Why Software Engineering? (3)

- This has also allowed different physics to be coupled, namely *multi-physics*, e.g. hydrodynamics, radiation transport, fluid-structure interaction;
- Data structures are subsequently different depending on the scale and the calculations operating on them;
- Each of the individual physics are packaged and need to interact with other physics packages via interfaces (APIs).



Why Software Engineering? (4)

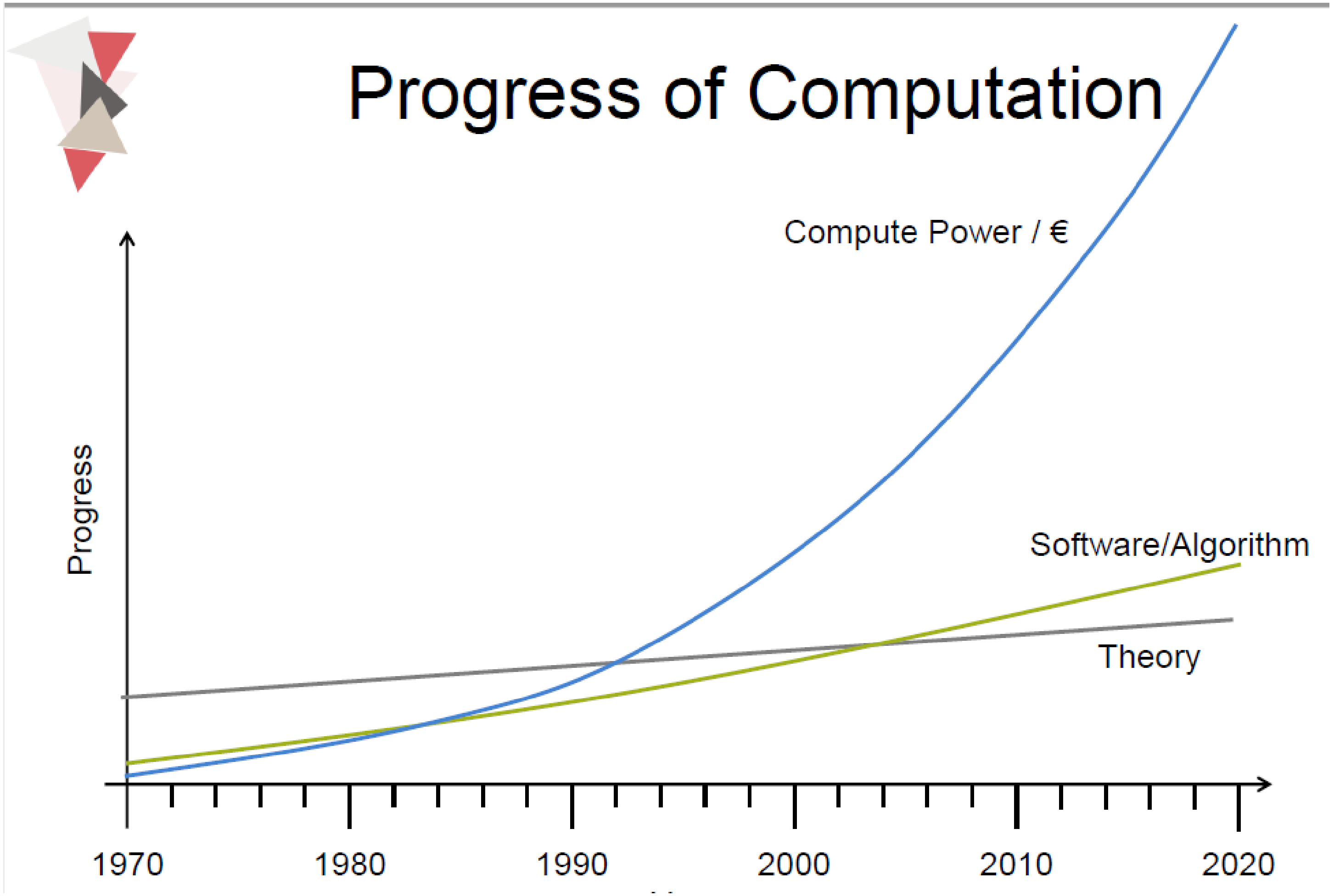
- Scientific applications usually have multiple layers [1]:



- The higher up the layer you develop, the more relevant software engineering is particularly at the "full application unit" and above.

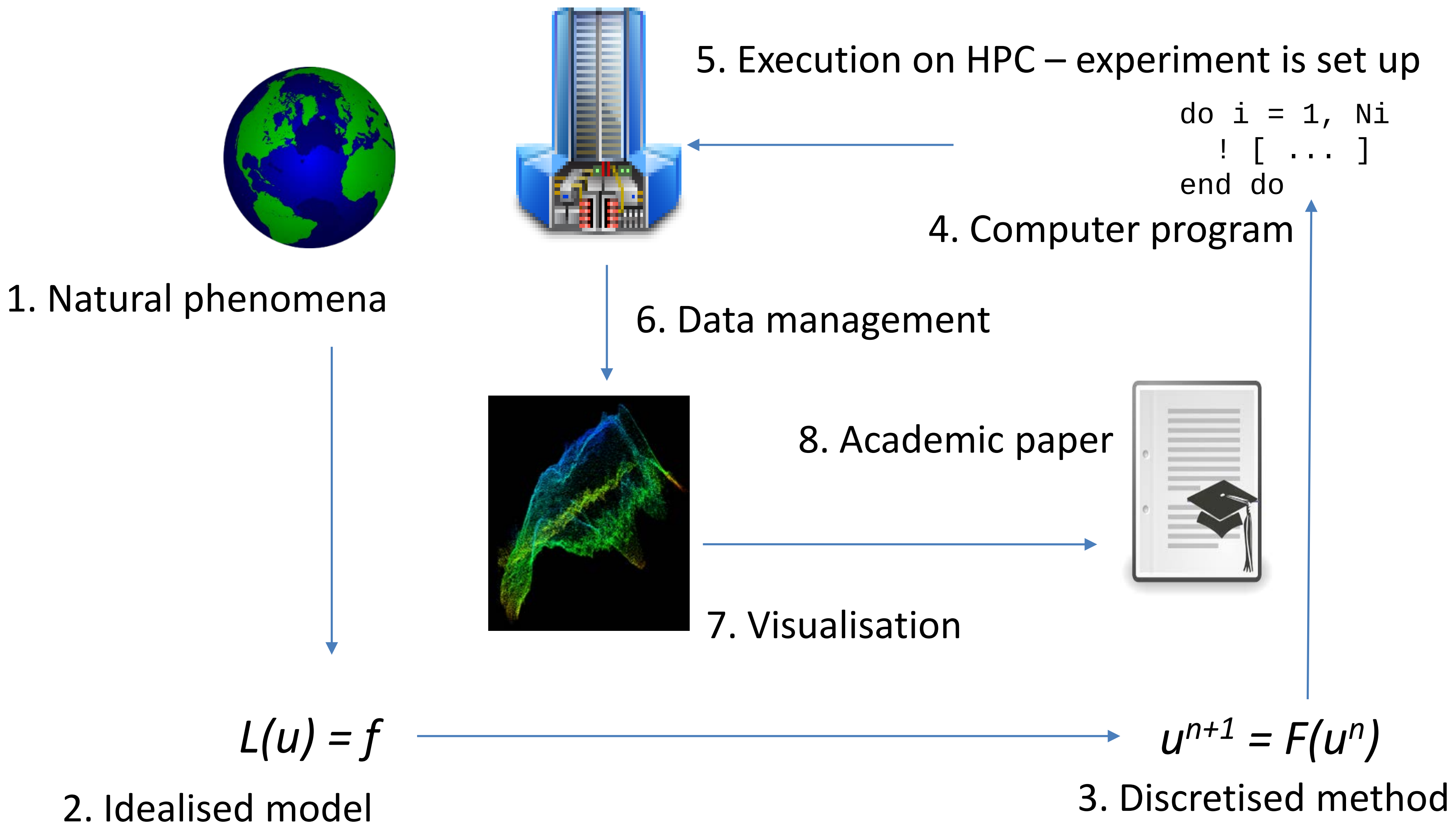
[1] "On the Role of Co-Design in High Performance Computing", R. Barrett, et al. Transition of HPC Towards Exascale Computing. 2013

Progression of Computational Software

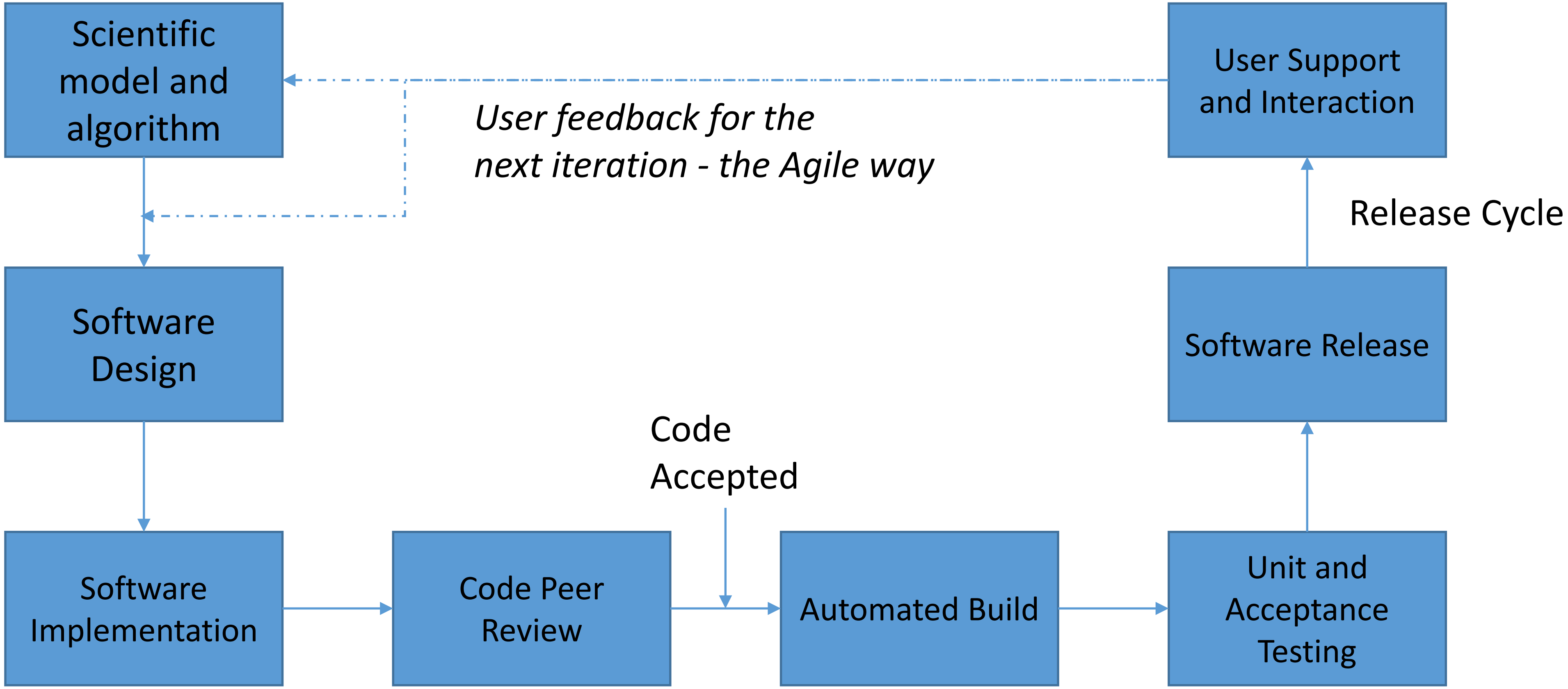


"Practical Use of Atomistic-Scale in Materials Engineering", Xavier Rozanska and Brian Dron. NAFEMS UK Conference. 2016

Computational Science Workflow

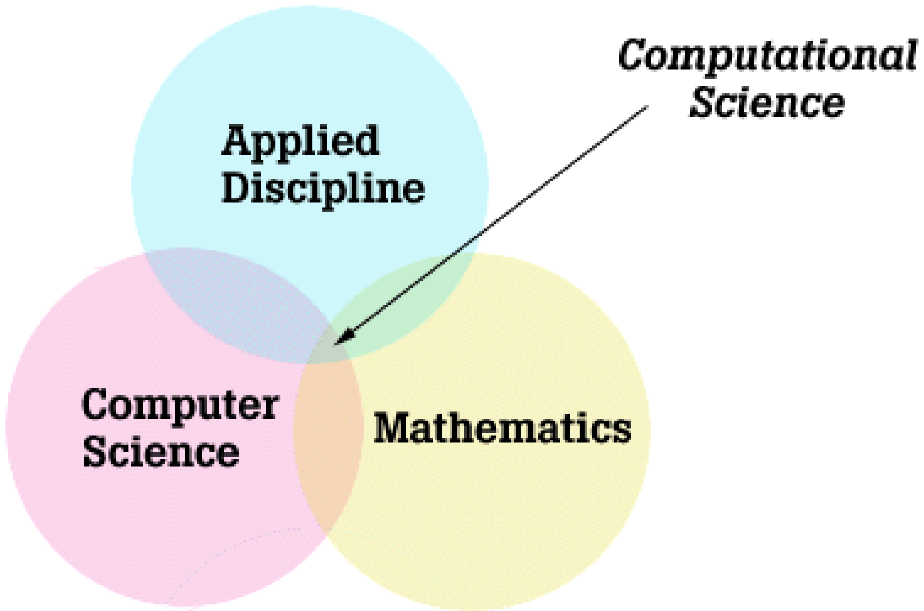


Software Development Workflow



Uniqueness of Computational Software (1)

- Testing computational codes are not easy;
- Verification and *validation* is difficult. Solutions are not always known as scientists are exploring new areas of science;
- Validation is ensuring models, e.g. differential equations, accurately represent the scientific phenomenon of interest;
- Computational codes need to be *efficient* and are a *collaborative* effort;
- You are writing software for your community so make sure they can understand your code*;



*You are not writing code for the "average" person!

Uniqueness of Computational Software (2)

- Operates on floating point data types which can cause numerical errors;
- Numerical algorithms can become unstable although algorithmically they look correct;
- Iterative algorithms can also diverge within a certain tolerance;
- Data usually requires some sort of post-processing, e.g. visualisation. Gaining new scientific insight from visualisation can also be difficult;
- This workshop will focus on the *computer science* skill set with the aim of making you an even better computational scientist!

Characteristics of Computational Software (1)

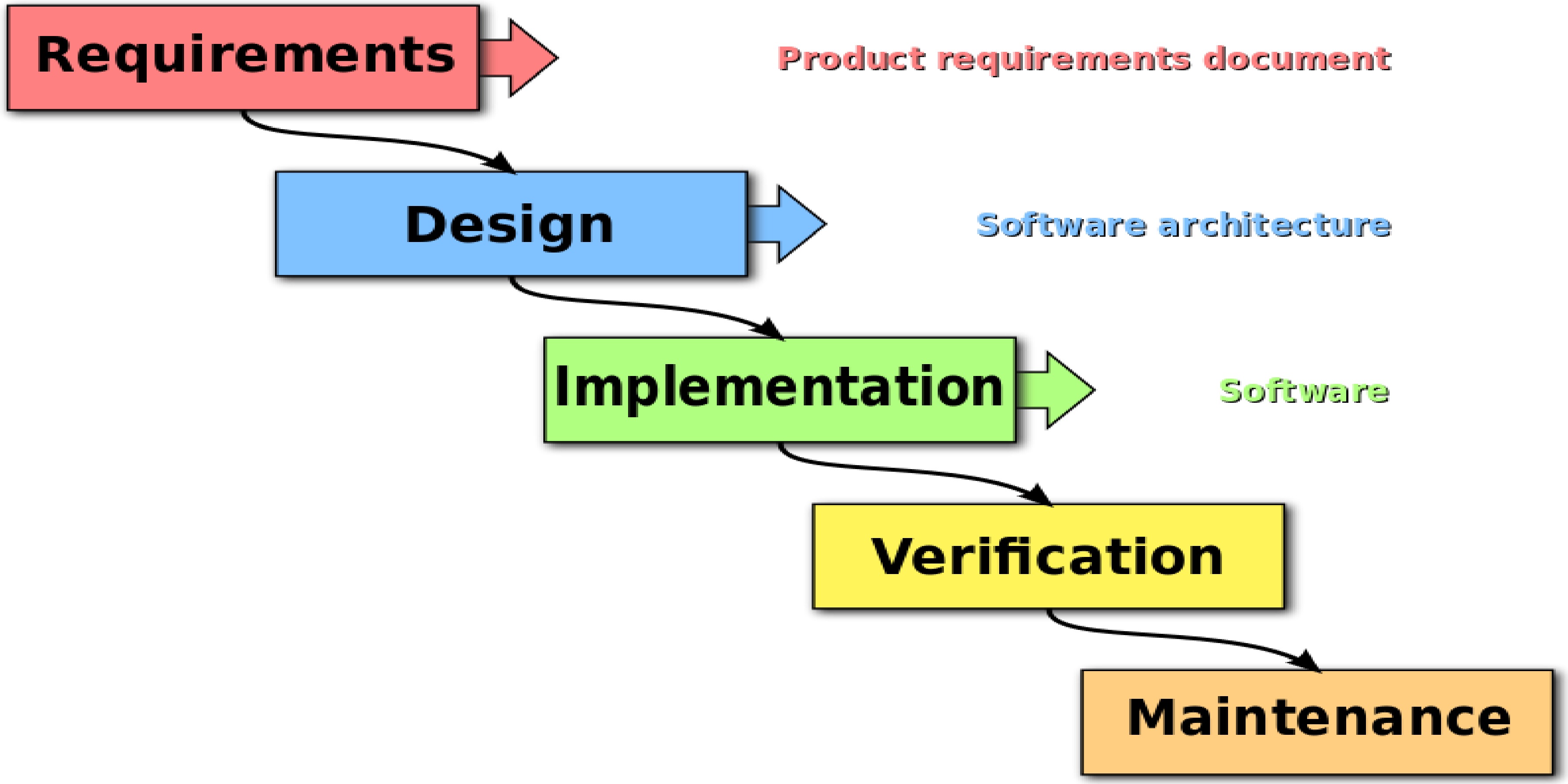
- Correct - it has to correctly implement the numerical algorithm;
- Portability - ideally it has to be *portable to other compilers and hardware architectures*;
- Testability - you have to be able to test it (within certain bounds) to determine its quality;
- Reliability - how *fault resilient* is your parallel code when a node fails?
- Performance - does it make efficient use of the hardware, e.g. scalability;
- Debugging - how easily can the code be debugged?
- Profiling - how easily can the code be profiled?

Characteristics of Computational Software (2)

- Maintainability - how easily the code can be maintained and changed, namely how extensible is it? *How well does it scale with increasing number of program units, e.g. Fortran modules?*
- Functionality - what functionality does it provide, e.g. solvers;
- ***Performance* and *portability* [1] are usually mutually exclusive, but it is possible to write performance portable code using the Fortran language standards** and this is one of the aims of this workshop;
- Portable codes avoid using language extensions, compiler specific features and can build using all compilers that all adhere to the Fortran standard.

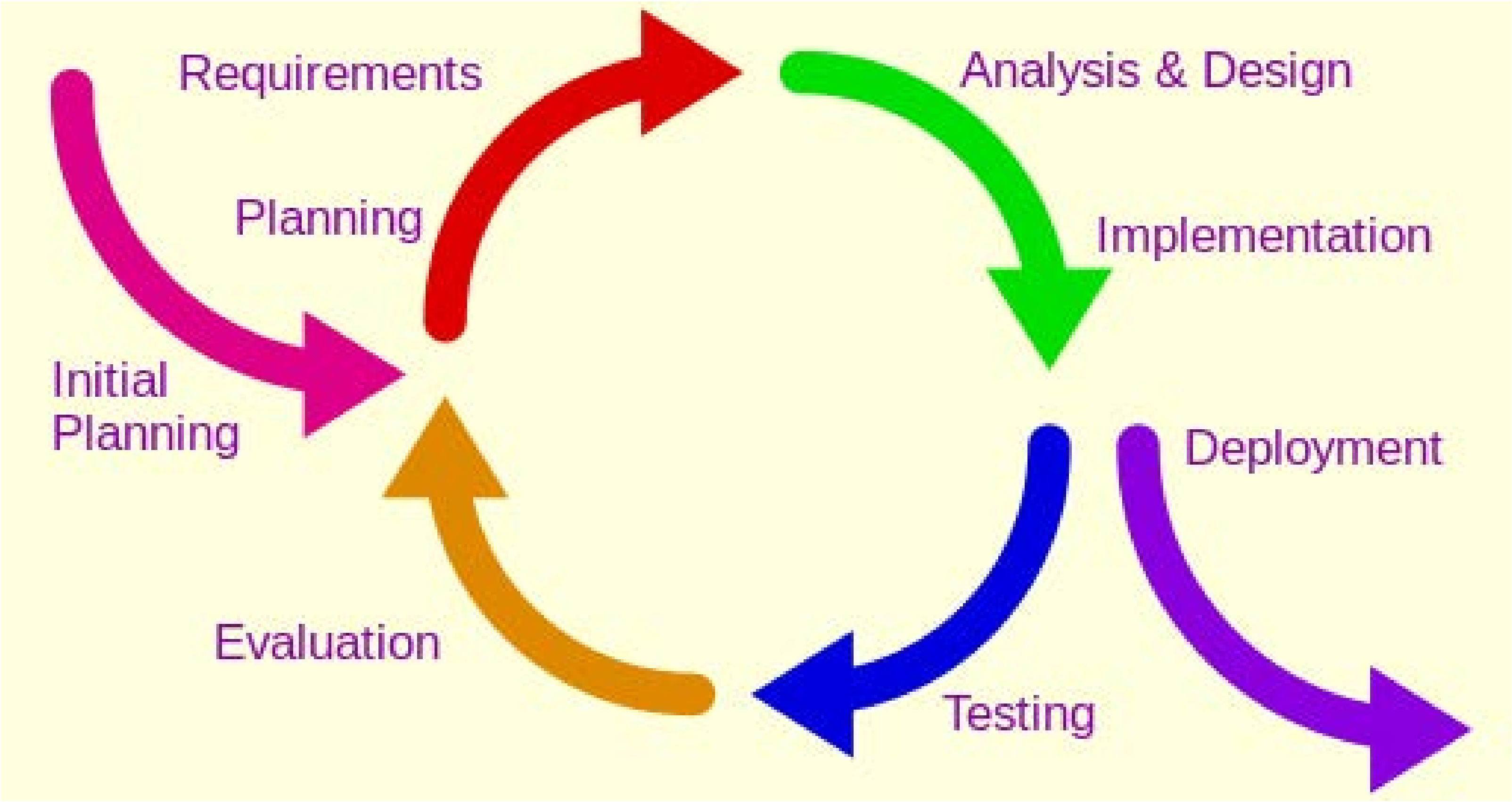
[1] http://www.hpcwire.com/2016/04/19/compilers-makes-performance-portable/

Waterfall Development Model



https://en.wikipedia.org/wiki/Waterfall_model

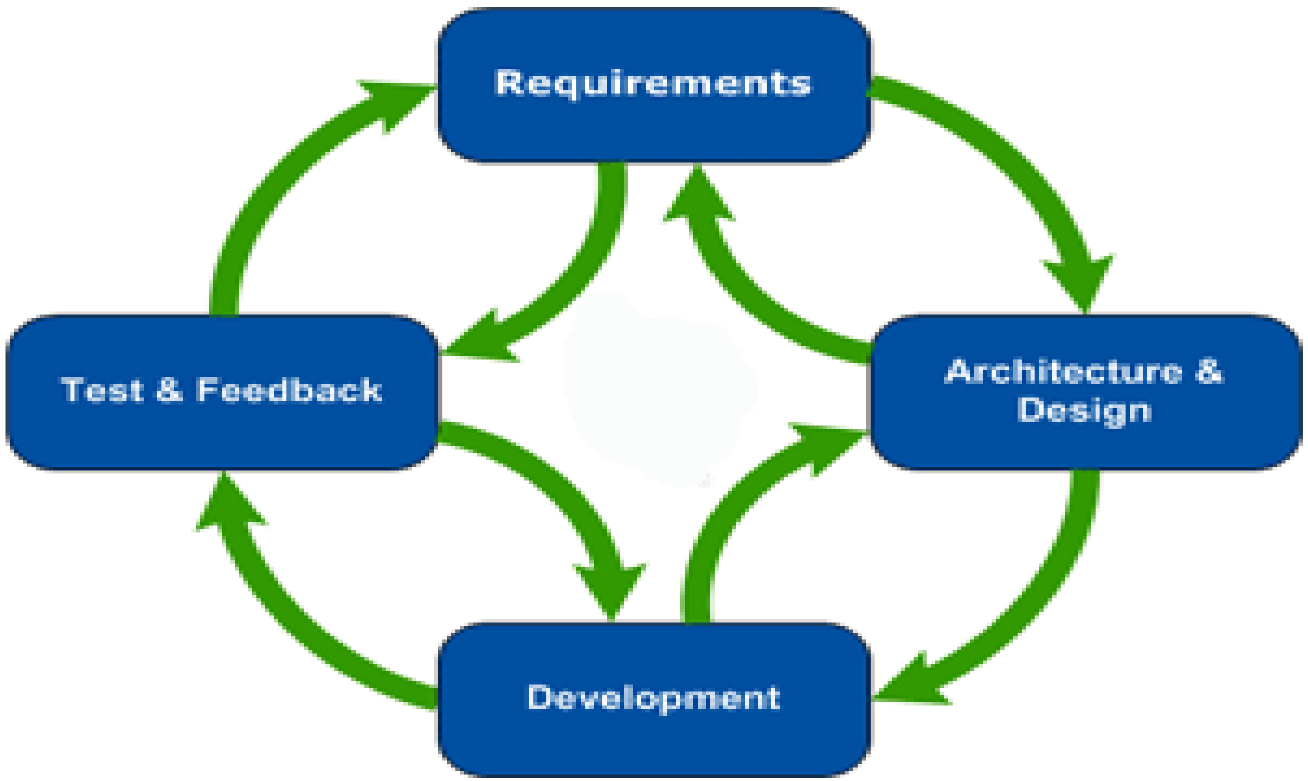
Iterative Development Model



https://en.wikipedia.org/wiki/Iterative_and_incremental_development

Agile Development Model (1)

- Individuals and interactions over processes and tools;
- Working software over comprehensive documentation;
- Customer collaboration over contract negotiation;
- Responding to change over following a plan;
- Incremental delivery of software features in *sprints*;



http://www.agilemanifesto.org/

Agile Development Model (2)

- Customer satisfaction by early and continuous delivery of useful software at *every release cycle*;
- *Welcome changing requirements, even late in development;*
- Working software is delivered frequently by communicating with users;
- *Close, daily cooperation between scientists and developers;*
- Face-to-face conversation is the best form of communication;
- Working software is the principal measure of progress;
- Self-organising teams;
- Regular adaptation to changing circumstance.

http://www.agilemanifesto.org/

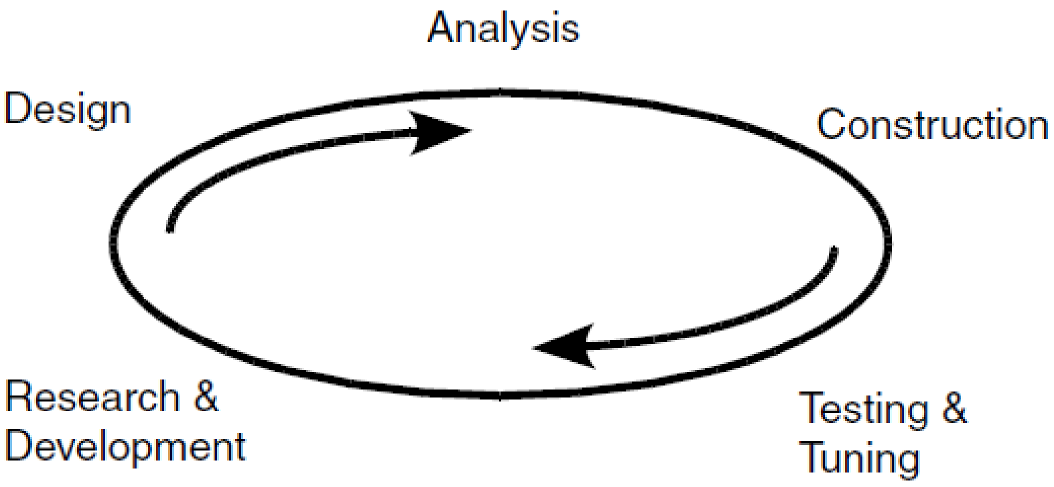
Development Model for Science (1)

- *Waterfall* model is out-dated and is difficult to adapt to the pace of science and engineering;
- *Iterative* model is more suitable for science and engineering as it can quickly adapt to changing needs;
- *Agile* is a highly flexible and collaborative model and can be used to test new hypotheses and models quickly;
- Agile methodology is useful for exploring new numerical algorithms, parallel models and computer architectures for your domain;
- Agile encourages a collaborative approach to software development: domain scientists, numerical analysts and computer scientists. This is also known as CoDesign [1];

[1] https://codesign.llnl.gov/

Development Model for Science (2)

- Scientific software lifecycle described by [1] is shown below:



- This correlates very closely to the Agile model;
- Which model is right for you? This will depend on your own scientific domain, but the Agile model does look very promising.

[1] "Scientific Software Design", D. Rouson. Cambridge University Press, 2014.

Software Management Portal

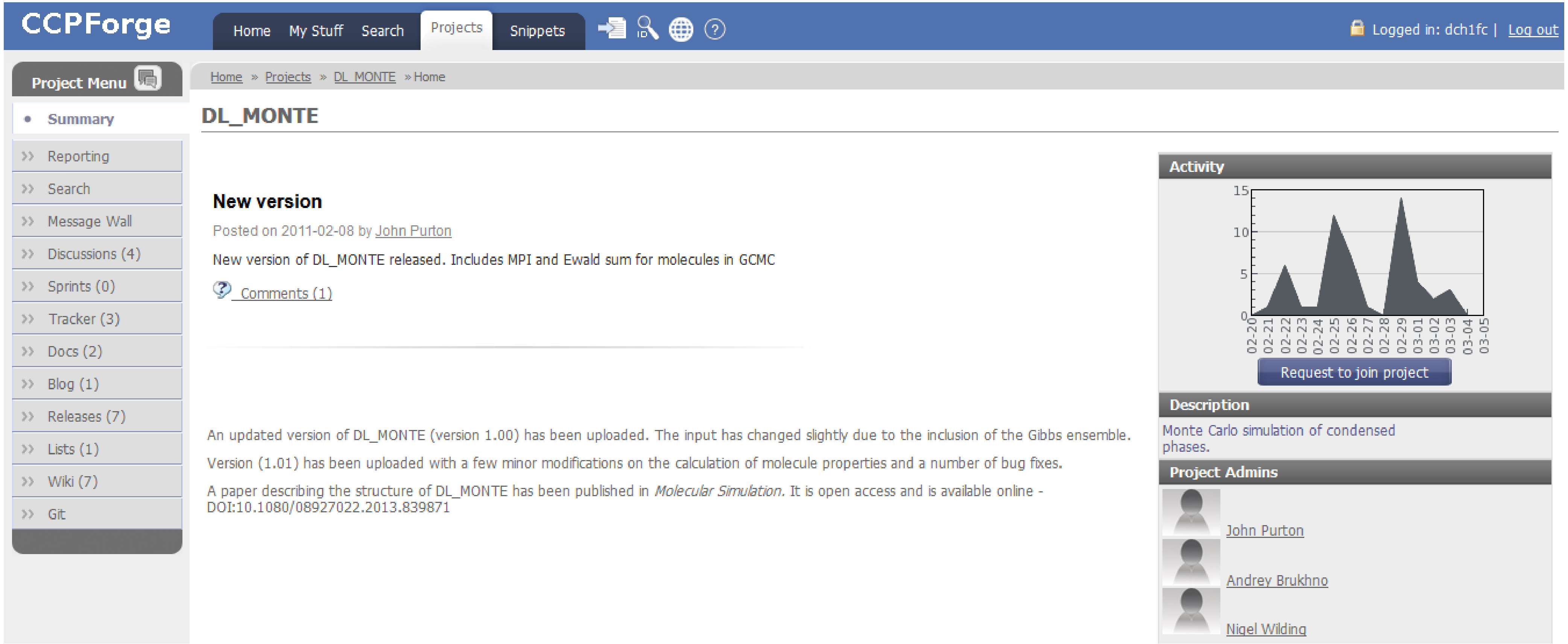
- Every code should have its own web portal where the code, e.g. an issue tracking system;
- The portal manages code as a project and progress is monitored by developers;
- It also hosts the central code repository which is read only viewable;
- All issues/bugs are tracked and monitored on the portal which are stored in a central database;
- Code documentation is made available as well as source code for users;
- Announcements are made on the portal and also through mailing list of users;
- This enables collaborative working for a geographically dispersed team;
- A number of open source portals are available, e.g. CCPForge [1].

[1] https://ccpforge.cse.rl.ac.uk

CCPForge - Portal for Computational Codes

- Provides a discussion forum for users;
- Issue tracking and, task and project management;
- Document management;
- Software project announcements to users including mailing list management;
- Wiki functionality for documentation;
- BuildBot continuous integration feature which performs automated builds of your code. Code release management;
- Version control integration with Git, SVN, Mercurial and CVS.

CCPForge Web Interface



Object Oriented Programming (OOP)

- OOP has been popularised by the C++ language and is widely used within computer science;
- OOP is provided by the Fortran 2003 standard and can be emulated in Fortran 90 [1];
- Encapsulation – grouping data and operations on the data into a single object, e.g. vector or matrix;
- Inheritance – allows one object to acquire the properties of another object to create a hierarchy. This allows code re-use and extensibility;
- Polymorphism – a single operation applied to different data types;
- OOP is being slowly accepted within computational science because of its ability to aid good software engineering for *large* codes.

[1] "ExpressOOPFortran90.pdf" file on web link

OOP in Computational Science

- Adoption of OOP in computational science is limited due to the high computational cost of extra language features;
- OOP is mainly used in very large scientific codes, e.g. AMBER (molecular dynamics), IFS (weather modelling);
- If OOP is used for computational science, its performance degrading features should be outside compute intensive loops;
- Should you use OOP for your codes? If you do, use it carefully. It is mainly beneficial for very large codes, e.g. multi-physics and multi-scale;
- Exploit its SE benefits and avoid the performance penalties of OOP;
- For performance, keep data types as simple as possible, e.g. arrays.

Software Design

- Abstraction - allows functionality to be hidden from users and provides features via the Fortran use statement. Features that are logically related are grouped together in a form of an abstraction and features are exposed via APIs;
- Encapsulation - data structures that are relevant to a feature are also hidden and complexity hidden away from users. Relevant data structures are accessed via *get* and *set* subroutines;
- Modularisation - codes are decomposed of modules which make it easier to manage, particularly for large codes;
- Hierarchy - modules and components are arranged in a logically hierarchical manner.

Abstraction for Computational Science

- Kernels - sparse and dense linear algebra. Linear and non-linear solvers. *Catalogues of libraries exists for kernels so please check if they exist for your needs*, e.g. Intel MKL, NAG Library, GNU Scientific Library;
- Skeleton codes - just contains the communication if code is parallelised. See PRACE CodeVault [1] for examples;
- Compact or mini applications - connects the kernels and skeleton codes to do basic science;
- The code one develops should use the above building blocks and connects them together in a systematic manner;
- *Do not use Fortran I/O for data* - use NetCDF or HDF5 which also have parallel (MPI) implementations;
- Keep your data structures as simple as possible, e.g. arrays.

[1] http://www.prace-ri.eu/prace-codevault/

New Codes: Parallelism from the Beginning?

- Multi-core architectures such as CPUs and GPUs are so prevalent that it makes obvious sense to exploit them;
- This then poses the question of whether one should parallelise codes developed from scratch. *The answer is yes if you can*;
- If you are writing sequential code from the beginning, write it with parallelism in mind so that it can be parallelised in the future;
- Refactoring/parallelising at a later stage usually costs more, so it is better to parallelise from the beginning;
- Access to more memory is another reason for parallelism;
- Shared memory or distributed memory parallelism? Distributed memory parallelism provides the most performance and flexibility, and provides the best I/O throughput.

Software Abstraction

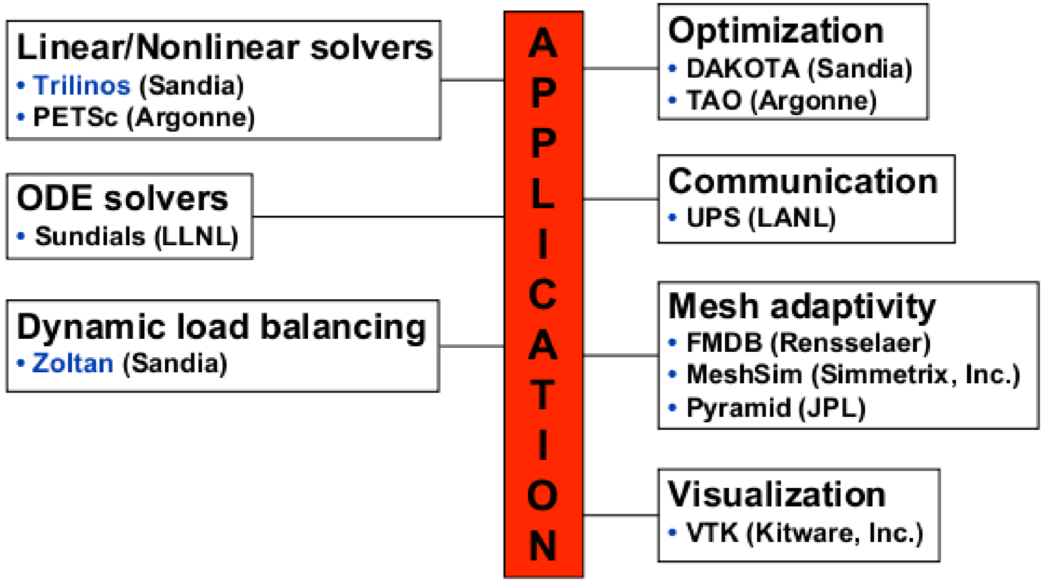
- Functionality can be grouped together based on types of operations, e.g. numerical linear algebra, non-linear solvers, discretisation;
- This allows such functionality to be used within multiple disciplines;
- Functionality can be grouped together based on a sub-domain, e.g. CFD, particle physics;
- Communication and parallel I/O layers for sub-domains, e.g. GCOM and H5hut;
- Multi-physics coupling between packages, e.g. CCSM;
- Different language bindings, e.g. calling Fortran from Python;
- The commonality can be grouped together to form a *component*.

Application Modularisation (1)

- Break down your code into smaller parts using Fortran modules and add them to a *library*;
- Modules will interact with each other via application programming interfaces (APIs) which need to be properly designed;
- Modules offer data abstraction features;
- Individual developers can work on their own module which avoids conflicts with other developers, thus avoid merge conflicts in version control systems, e.g. Git;
- Modules allow software reuse which can be used for a different code and *even in a different programming language*;
- Modules should be able to be *plugged* into any code;

Application Modularisation (2)

- Much easier to manage than a large monolithic code;
- Modules that have changed only need to be re-compiled and not the entire code;
- Modules should be developed such that they can be tested (verification).



http://www.cca-forum.org/

Software Engineering and Exascale

- HPC is heading towards Exascale computing and it is the large multi-physics and multi-scale codes that can exploit such machines;
- Single-scale and single-physics have less of a need for Exascale;
- For multi-scale multi-physics codes to exploit Exascale machines, additional physics packages are coupled to an existing code base;
- However, this requires the code to scale to Exascale as well as *software scalability*. How easily can the code base be scaled?
- Hence, the need for good software engineering for computational science.

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 2015 improved the CoArray features by adding collective subroutines, teams of images, listing failed images and atomic intrinsic subroutines – 2015;
- Most compilers, to date, support Fortran 77 to Fortran 2008. See [1] and [2] for further details;
- This workshop will be mainly discussing Fortran 90, 2003 and 2008.

[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, PathScale and Absoft;
- Fortran compiler vendors then implement the agreed standard;
- Some vendors are quicker than others in implementing the Fortran standard;
- 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

Codes Developed in Fortran

- Chemistry: Gromacs, AMBER, CASTEP, DL_POLY, DL_MESO, Gaussian, CP2K, GAMESS, Quantum Espresso, VASP and Tinker;
- Weather simulation: IFS, WRF, GungHo;
- Computational Fluid Dynamics: StarCD, CFX, AVL Fire, COMSOL
- Manufacturing: LS-DYNA, Abaqus, Ansys;
- Oil and gas: Schlumberger Eclipse;
- Some R and Python linear algebra kernels are written in Fortran;
- Many dense linear algebra libraries developed in Fortran, e.g. BLAS, LAPACK, Scalapack.

Fortran Usage on Archer[1] HPC Service (1)

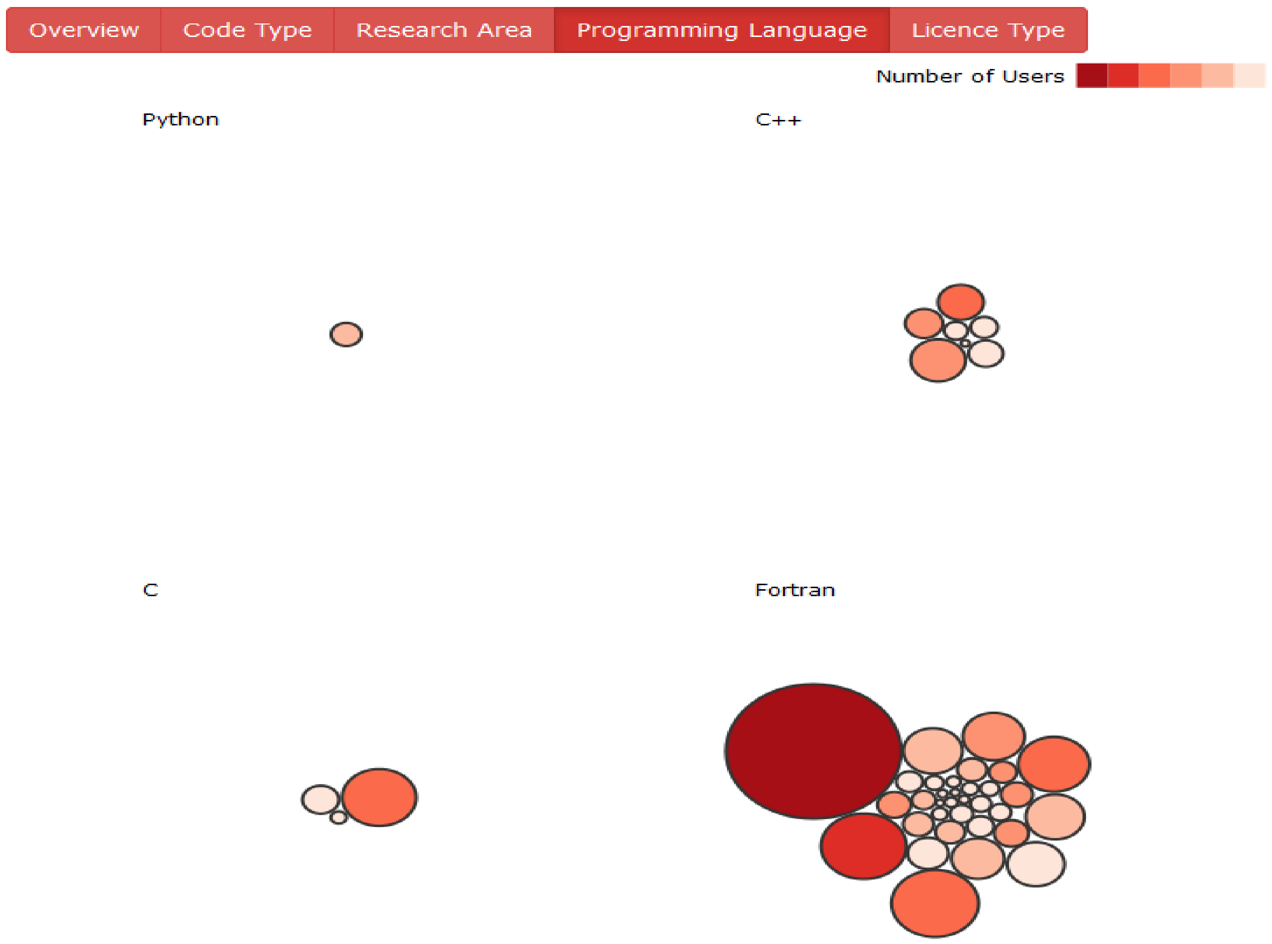
- Programming language usage is:

Programming Language	%Time	%Jobs
Fortran	**73.58**	**80.03**
C++	7.44	4.74
C	7.4	3.14
Python	0.67	0.99
Others	14.21	11.75

- Fortran usage statistics is ≈ 70% for the Top 500 supercomputers [2];
- *Fortran is the dominant programming language of computational science and engineering*.

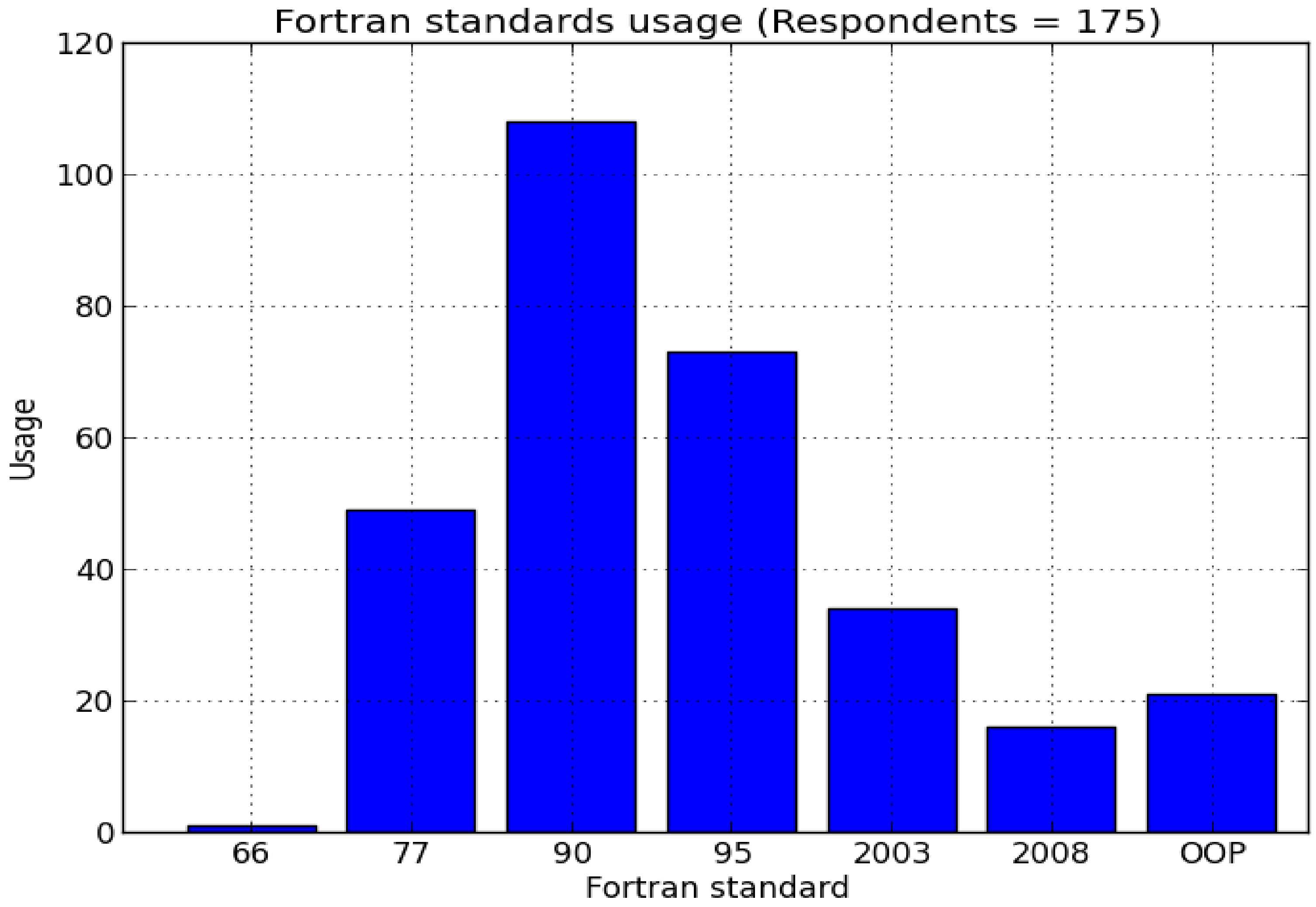
[1]The UK National Supercomputing Service, ARCHER. www.archer.ac.uk
[2] www.top500.org

Fortran Usage on Archer HPC Service (2)



http://www.archer.ac.uk/status/codes/

Current Fortran Usage



Unique Benefits of Fortran (1)

- The *array* is the most common data structure in computational science. *Fortran array operations are faster than C/C++ and are more likely to be vectorised by the compiler*;
- Dynamic arrays in Fortran are not pointers, where in C/C++ they are pointers making them more difficult to deal with;
- In C/C++ a pointer to a pointer dereferencing (a two-dimensional array) requires two memory accesses, whereas in Fortran it is one;
- Has a rich history in computational science and has added many modern features of programming into newer standards;
- It is keeping pace with modern software development and computational science;
- The Fortran standard (and compilers) still support Fortran 77 so your legacy code can still run;

Unique Benefits of Fortran (2)

- *It is the only language standard that provides distributed memory parallelism via CoArrays (Fortran 2008);*
- It is not used in any other domain area, e.g. writing operating systems, web development, databases. *It is not a general purpose programming language* like C, C++ and Python;
- It is a language that has been designed *exclusively for numerical computation* and has applications only in computational science and engineering;
- A blog is written by Steve Lionel of Intel (also known as “Doctor Fortran”) which covers modern Fortran [1]

[1] https://software.intel.com/en-us/blogs/author/512685

Deleted and Obsolescent

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

Real and double precision DO variables

Deleted

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

Alternative

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

- Use integers

Branching to an END IF statement from outside its block

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

Deleted

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

Alternative

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

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

PAUSE statement

- Suspends execution

Deleted

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

Alternative

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

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

ASSIGN and assigned GO TO statements and assigned FORMAT specifiers

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

Deleted

```
    assign 100 to lbl
    ...
    go to lbl[[,] (label-list)]
    ...
100 continue
    ...
    assign 200 to fmt
    ...
    print fmt, 27
    ...
200 format(i4)
```

Alternative

```
    lbl = 100
    ...
    if (lbl == 100) go to 100[[,] (label-list)]
    ...
100 continue
    ...
    fmt = "(i4)"
    ...
    print fmt, 27
```

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*)

Obsolescent

```
    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

Obsolescent

```
    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

Alternate return

Obsolescent

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

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

- Use integer return with IF or SELECT CASE construct

Alternative

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

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

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

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

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

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

- The difference between function and array references are clear;
- Capitalise names of constants:

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

[1]Exceptions 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:

```
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 in the file. 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;
- Loops variables such as `i`, `j`, `k`, `l`, `m`, `n` are fine to use as they are routinely used to describe mathematical algorithms;
- Reflect the variables as much as possible to the equations being solved; so for $p = \rho RT$:

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

- Above is an example of self-describing code;
- 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
```

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 reference[1];

[1]Consult 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 vectorisation/optimisation reports;
- Avoid multi-dimensional arrays where possible. Use one dimensional arrays if you can as it is more efficient to access array elements;
- Try to use array sizes that are powers of 2 which makes it easier for the compiler to index.

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)

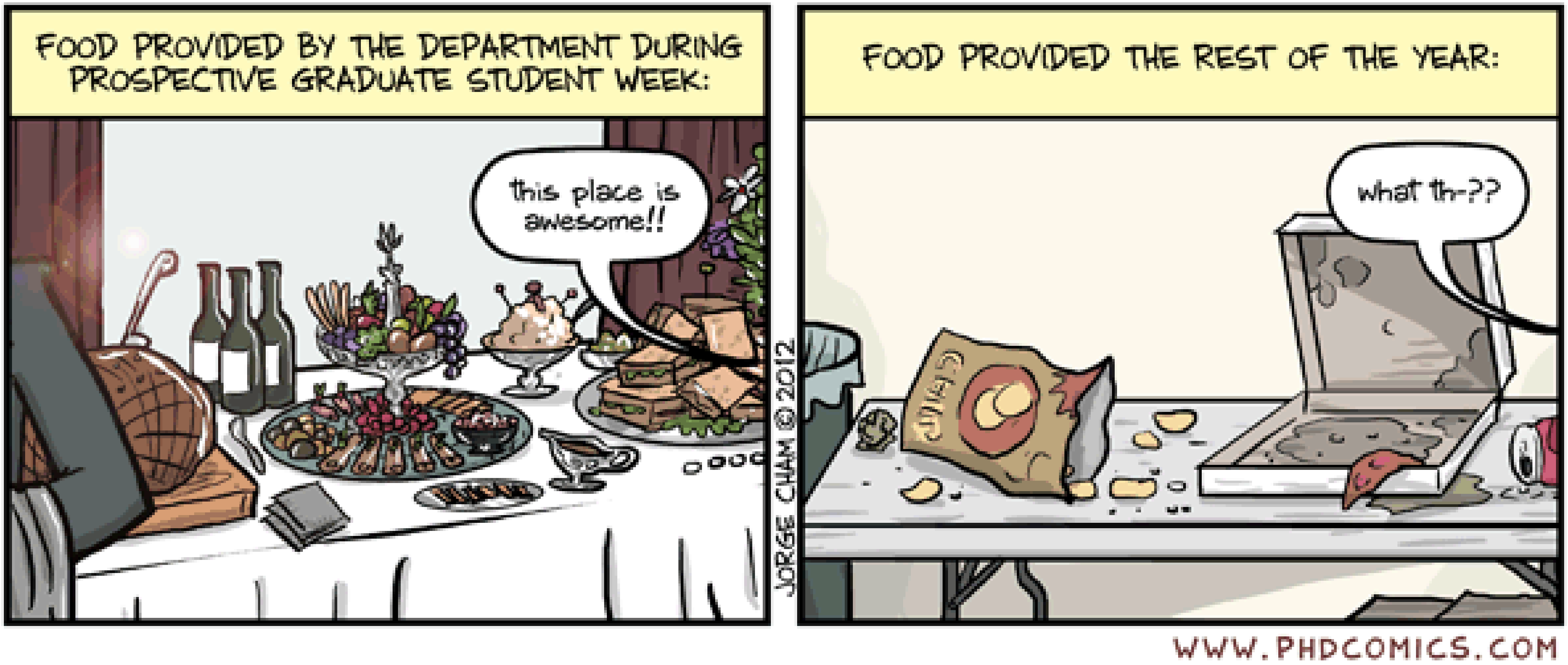
- For pointers, use the `p` suffix:

```
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
```

Lunch Time



FOOD PROVIDED BY THE DEPARTMENT DURING PROSPECTIVE GRADUATE STUDENT WEEK:
this place is awesome!!
JORGE CHAM © 2012
FOOD PROVIDED THE REST OF THE YEAR:
what th-??
WWW.PHDCOMICS.COM

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 by the compiler which will help you write correct code.

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, functions and subroutines 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;
- Use LaTeX syntax if required. Source code documenting systems such as Doxygen 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;
- 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 integer `ierr` is zero if allocation is successful. If this is non-zero, then check the error message variable `msg`;

[1]Use Valgrid or RougeWave MemoryScape to debug memory problems

Memory Management (2)

- Then deallocate when not required:

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

- Remember to deallocate – if not, it could cause memory leaks[1];
- Can use the `allocated( array )` intrinsic function to check whether memory has been allocated.

[1]Use Valgrid or RougeWave MemoryScape to debug memory problems

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 90 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), pointer :: 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 in the compilation process;

```
#ifdef DEBUG
  print *, ‘count is’, counter
#endif
```

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

```
ifort -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`;
- 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;
- If using a different standard, other file extensions are also accepted: `.f95`, `.f03` and `.f08`. The pre-processed versions are `.F95`, `.F03` and `.F08`, respectively.

Numerical Data 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
gamma = 1.45E-10_DP
```

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

Numerical Data Types (2)

- 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

IEEE Floating Point Arithmetic

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

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

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
```

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

- 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*;
- Do not use external procedures! Always encapsulate them using Fortran modules.

Basic Polymorphism in Modules

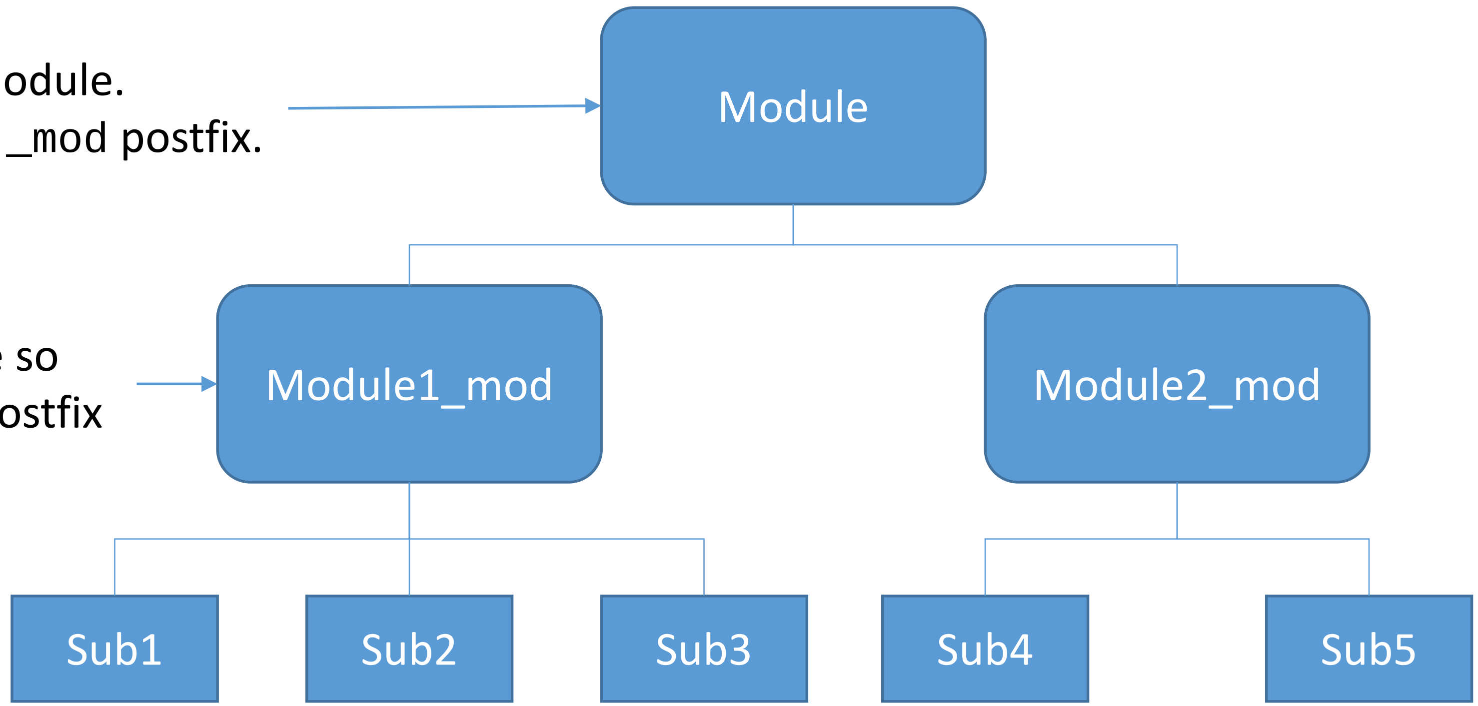
```
module vector_mod
  interface my_sum
    module procedure real_sum
    module procedure int_sum
  end interface
contains
  function real_sum( vec )
    real, intent(in) :: vec(:)
  end function real_sum
  function int_sum( vec )
    integer, intent(in) :: vec(:)
  end function int_sum
end module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

```
program main_prog
  use vector_mod

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

  print *, my_sum( veci )
  print *, my_sum( vecr )

end program main_prog
```

Fortran Module Hierarchy

Public facing module.
Do not put the _mod postfix.

Internal module so
it needs _mod postfix

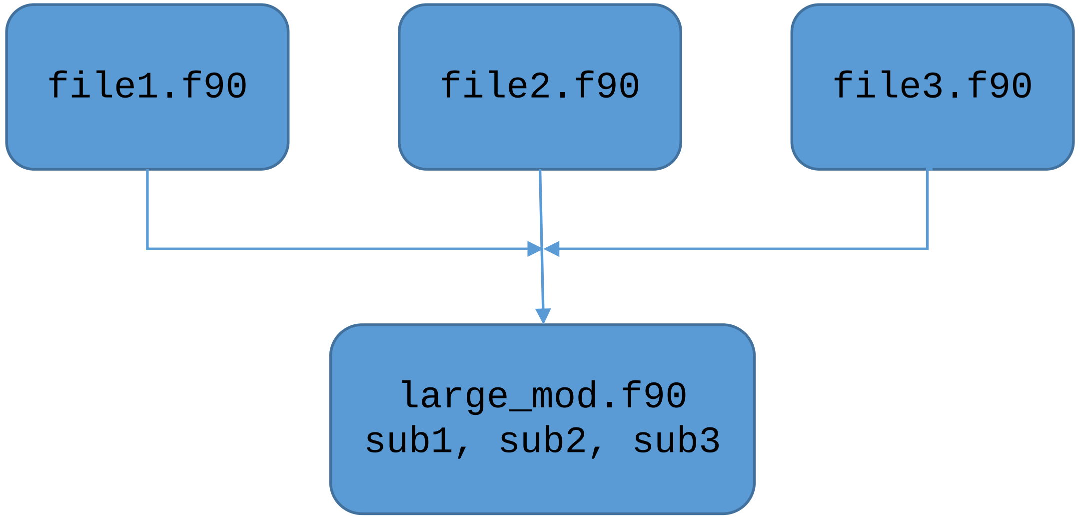


Fortran Submodules (1)

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

Good API Characteristics (1)

- 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;
- Ensure subroutines contained with your API are consistent which makes them easy to use and remember;

Good API Characteristics (2)

- Use Fortran modules for developing APIs as they check argument parameters at compile time;
- Removing functionality should be only be done collaboratively with users and after effective consultation/communication of the API user community;
- Changing APIs will break existing codes, so use the Fortran keyword `optional` to extend your existing API;
- Before changing or releasing your API, ensure code is reviewed;
- If API is parallel, ensure baseline parallel performance is recorded, i.e. strong and weak scaling. All new releases should at least provide this level of performance.

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

print *, sqr( [ 1.0, 2.0, 3.0 ] ) ! print 1.0, 4.0, 9.0
```

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.

Refactoring Legacy Fortran Codes (1)

- Refactoring is the process of changing the code structure to make it more readable and cleaner, but does not change its functionality;
- Over time, the code's integrity and design starts to decay, hence the need for refactoring;
- Ensure the code adheres to latest programming standards;
- Improves the design and maintainability of the code;
- Makes code easier to understand – for you and other developers;
- Helps you find bugs easier and quicker;

Refactoring Legacy Fortran Codes (2)

- Use refactoring tools to refactor language tokens. Medium to large refactoring needs to be done manually, e.g. changing blocks of code like loops and branches;
- After refactoring, you must test your code!
- *Every change can introduce a bug so you absolutely must version control your code so you can revert back to a working version!*

When to Refactor Code

- Refactor when you add new functionality;
- When you parallelise and/or optimise your code - a form of refactoring;
- Refactor when you fix a bug - it is a sign you need refactoring, because the code was not clear enough for you to see there was a bug;
- Refactor during code reviews;
- Mapping data types that match data structures in maths and science;
- When code is so broken and cost of refactoring is too high it might be better to start a brand new code. If more time is spent refactoring than adding features, it is a sign that you may need to redevelop your code;
- *Fortran 66/77 codes should be refactored*. If it is too complex to refactor, you may create Fortran 90 wrappers for it using the NAG Fortran compiler;
- Refactoring APIs will break codes, so API design should be done with care.

Types of Refactoring

- Abstraction - a layer for managing complexity;
- Encapsulation - collection of related data types and operations on them;
- Modularisation - separating functionality into different entities;
- Hierarchy - listing layers of abstraction in a certain order;
- Optimisation - changes to code to make it run faster which includes parallelisation;
- The above refactoring types must be done manually. No such tool can do this for you.

plusFORT SPAG

- Converts Fortran 66 and 77 to Fortran 90 and removes clutter;
- Adds `implicit none` and explicitly declares all variables;
- Removes common blocks and replaces them with Fortran modules;
- Renames short variables to more long meaningful names;
- Removes `goto` statements and replaces them with IF blocks;
- plusFORT SPAG is proprietary software;
- Eclipse Photran and CamFort do limited refactoring and are free.

http://www.fortran.uk/fortran-analysis-and-refactoring-with-plusfort/

NAG Fortran Compiler Polish (1)

- The NAG compiler has some refactoring features;

```
nagfor =polish code.f90 [options] -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}`;
- `-relational=X` - Specifies the form to use for relational operators, X must be either `F77-` (use `.EQ., .LE.`, etc.) or `F90+` (use `==, <=`, etc.).

Code Reviews

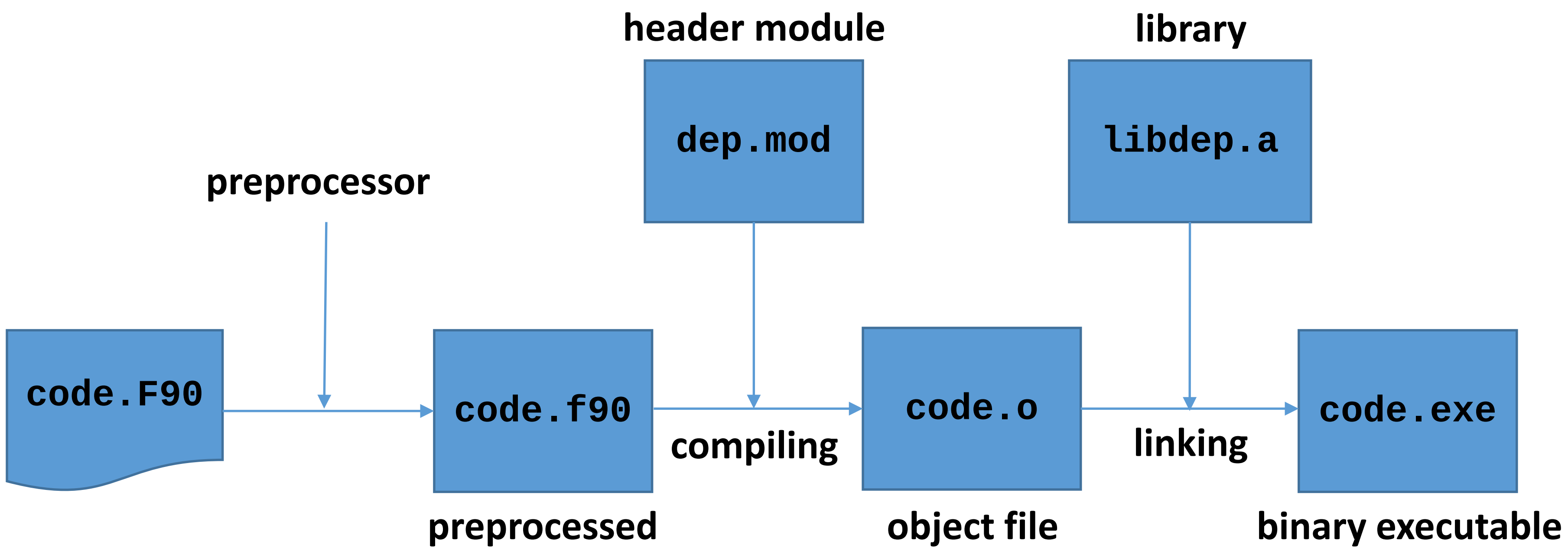
- Code reviews allow developers to periodically check the quality of the code;
- Ensures the code adheres to the group’s coding standards;
- Code reviews ensure the code is readable, portable and maintainable;
- It is testable and checks for any obvious bugs;
- Check that the code is adhering to the latest programming standard;
- First code review yourself by printing out your code;
- Then code review with a colleague or with your team;
- Do not review more than 400 lines of code in one hour sessions.

When Should You Code Review?

- It can be done during pair programming;
- Over the shoulder reviewing – explaining your code to a colleague;
- When there is a code commit using a version control system;
- Automated code reviews using tools;
- Codes are hitting a performance wall;
- New programming standards are released, e.g. Fortran 2015;
- When new numerical libraries are made available that provide more features and better performance;
- When new architectures are made available and corresponding libraries for the new architecture, e.g. GPUs.

“11-proven-practices-for-peer-review.pdf” file on NAG web page

Building of Codes



Build Commands

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

  ```
  ifort -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:

  ```
  ifort 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*;
- 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.

Ordering Libraries During Linking

- When linking multiple libraries with dependencies, the order of the libraries during linking is very important;
- Otherwise you will get the dreaded “undefined symbol” errors;

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

- The `netcdff` library calls subroutines from the `netcdf` library so *it must be listed in the above order*.

Creating Libraries

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

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

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

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

Compiling and Linking with Submodules

- Compile the module to create `large_mod.mod` and `large_mod.o`

```
ifort -c large_mod.f90
```

- Then compile the submodule to create `large_smod.o` and `large_mod@large_smod.smod`

```
ifort -c large_smod.f90
```

- Compile the main code to create `main.o`

```
ifort -c -I. main.f90
```

- Link the main code to create `main.exe`

```
ifort main.o large_smod.o -o main.exe
```

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 (95, 2003 and 2008) to ensure portability;
- Release 6.1 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 of the standard in order to modernise the language;
- NAG compiler documentation can be found at [1].

[1] http://www.nag.co.uk/nagware/np/r61_doc/index

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 just contain 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 via the Fortran `use` statement.

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

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

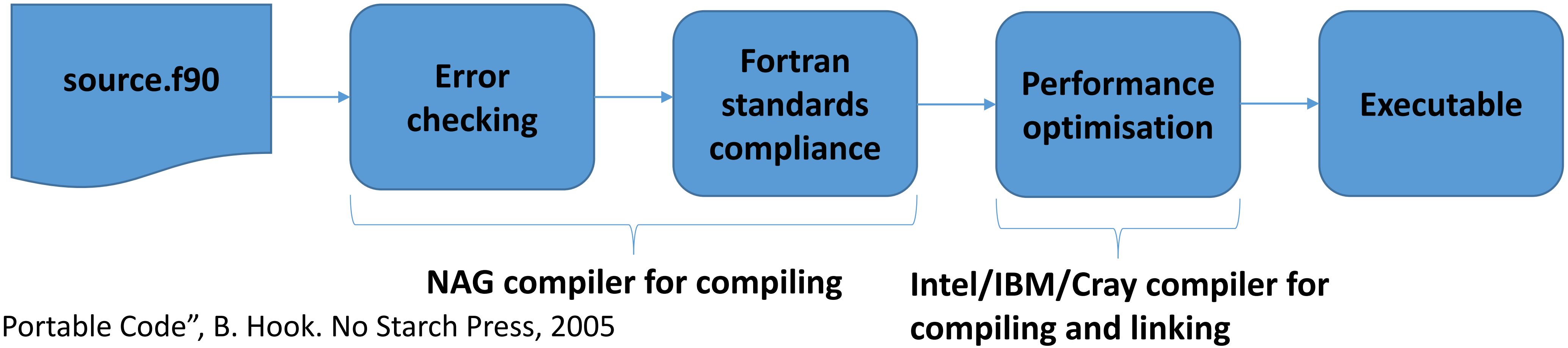
`-gline` - This flag will do a subroutine trace call when a runtime error has occurred;

`-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;

Writing Performance Portable Code

- 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 it 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 are 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
```

Parallel Builds Using Makefile

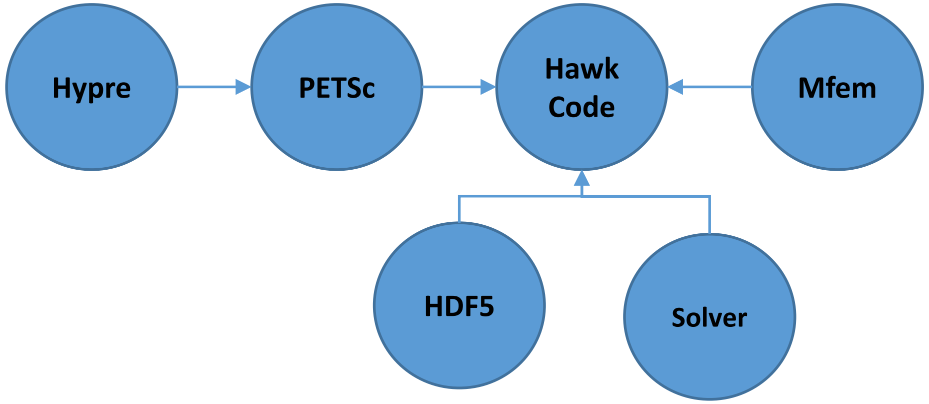
- When writing Makefiles, dependencies must obviously be correctly specified;
- If they are not, you will get link errors resulting in “undefined symbol” messages;
- In addition, parallel builds depend on rule dependencies being correctly defined and only then can you use parallelise builds;
- To parallelise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

```
make -j n
```

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

MixDown - Building Multiple Components

- Applications using the common component framework will have a number of components that need to be built for HPC systems;
- This can be tedious and time consuming;



- Use MixDown which connects components and builds a dependency graph. It then builds the individual components.

https://github.com/tepperly/MixDown

MixDown Component Configuration

`Name`: name of component, e.g. hawk;

`Path`: name of packages file, e.g. hawk.tar.gz;

`DependsOn`: component that hawk depends on, e.g. petsc;

`Fetch`: download library from the Web;

`Unpack`: unpack the component into a directory;

`Preconfig`: script that creates the configure script;

`Config`: configure script flags;

`Build`: build command;

`Install`: installation command;

`Clean`: command to clean the build;

Then execute `mixdown  hawk.md` to build the Hawk code and dependencies.

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*. Doxygen 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;
- Minimum documentation – the Agile way.

Doxygen Code Documentation

- Doxygen [1] is a code document generator that supports Fortran;
- It provides an easy way to navigate around the code base;
- Comments are parsed and form the documentation;
- Can parse LaTeX equations which are compiled and presented as images;
- Can output in HTML, PDF (using LaTeX) and RTF.

[1] http://doxygen.org/

Doxygen Configuration (1)

- Generate a template Doxygen file `doxygen -g code.dxg` in the `doc/` directory which will contain key-value pairs:

```
PROJECT_NAME             = "name of code"
PROJECT_NUMBER           = 3.0
PROJECT_BRIEF            = "short description of code"
OUTPUT_DIRECTORY         = doxygen
OPTIMIZE_FOR_FORTRAN     = YES
EXTENSION_MAPPING        = f90=FortranFree F90=FortranFree
```

Doxygen Configuration (2)

```
INPUT                   = ../src
FILE_PATTERNS           = *.f90 *.F90

GENERATE_HTML           = YES
GENERATE_LATEX          = YES
HAVE_DOT                = YES
CALL_GRAPH              = YES
EXTRACT_ALL             = YES
EXTRACT_PRIVATE         = YES
EXTRACT_STATIC          = YES
```

For `INPUT` always use a relative path

Comments for Doxygen (1)

- For a module add the following text:

```
! MODULE solver_mod
!> @author
!> Module Jane Smith, Bakersfield College
!> Description of the module
!> Solves \f$ \frac{d\lambda}{dt} = \frac{dz}{dt} \f$
```

- Will add the following equation into documentation:

$$\frac{d\lambda}{dt} = \frac{dz}{dt}$$

Comments for Doxygen (2)

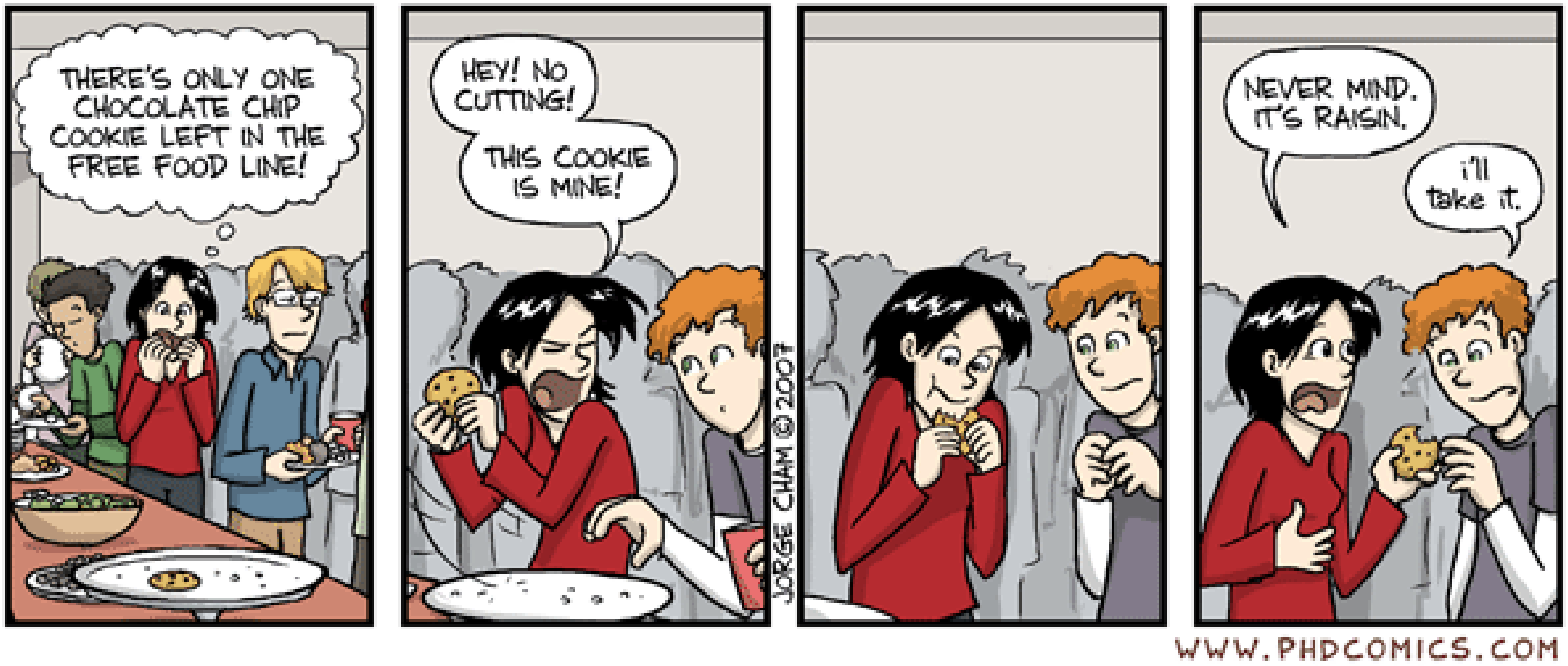
- For subroutines/functions, prefix comments and *use Doxygen comments for all arguments*:

```
!> subroutine that transposes a matrix
!> @param matT transposed matrix
subroutine TransposeMatrix( matT )
   real, dimension(:,:), intent(inout) :: matT
end subroutine TransposeMatrix
```

- For variables:

```
!> list of eigenvalues
real, dimension(:) :: eigs
```

Break followed by day one practicals



End of Day One – Practical 1

- To get the exercises, download them from the NAG web site:

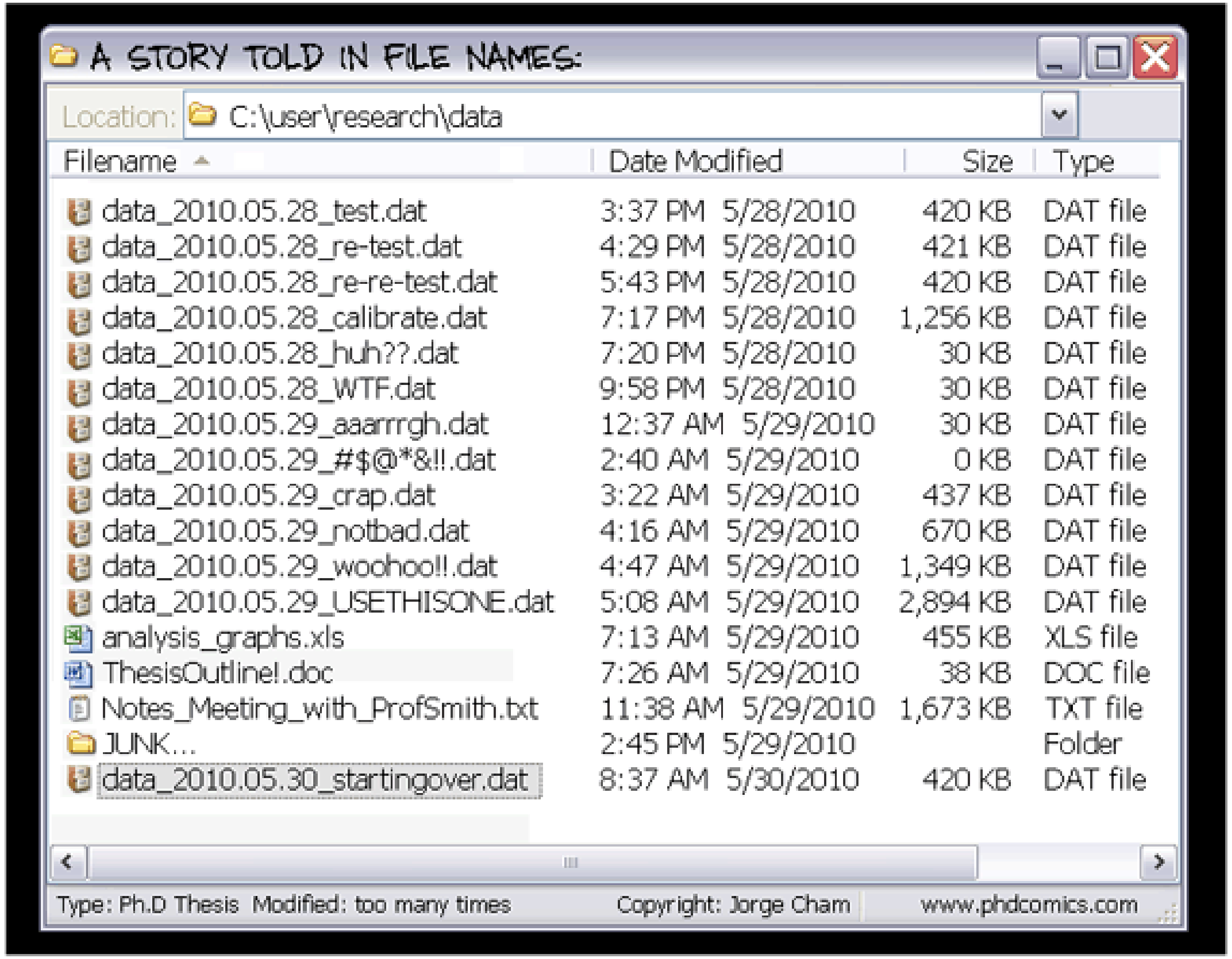
```
$ wget \
http://www.nag.co.uk/market/training/fortran-workshop/ex.tar
$ tar -xvf ex.tar
$ cd ex
```

- The exercises are listed in `exercises.pdf` or `exercises.ps`

Day Two Agenda

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

Data Management



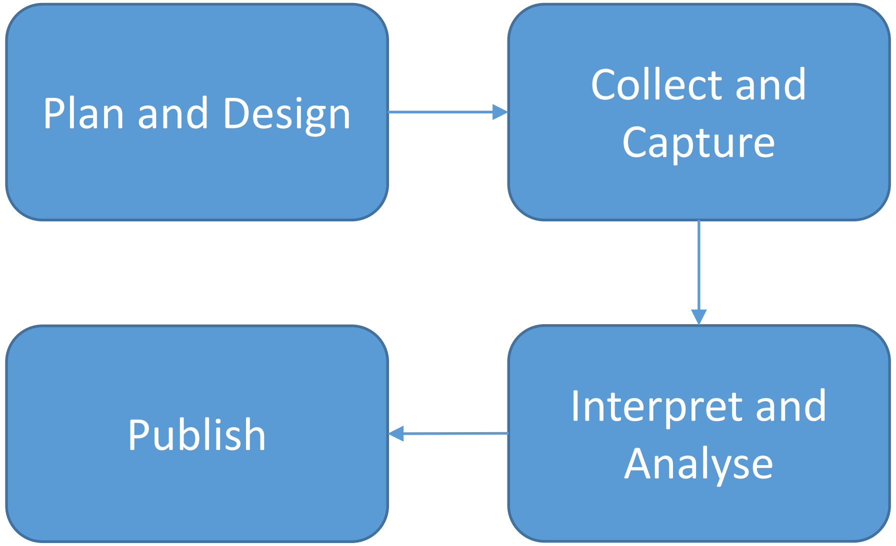
http://phdcomics.com

Data From Simulations

- Computational codes are producing petabytes of data from single to multiple simulations with various configurations 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

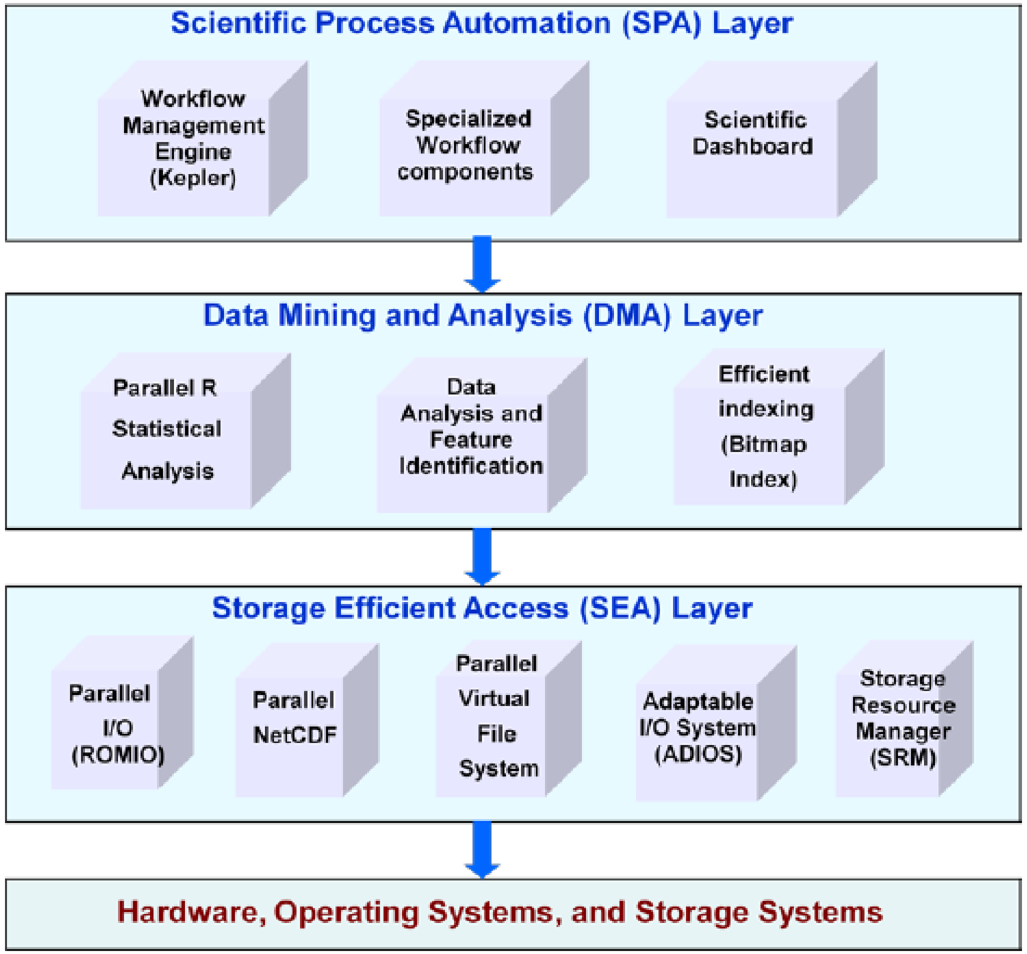
Challenges of Data Management

- 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;
- 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

Layers of Data Analysis



"Scientific Data Management", A. Shoshani and D. Rotem. Chapman and Hall, 2009.

NetCDF File Format

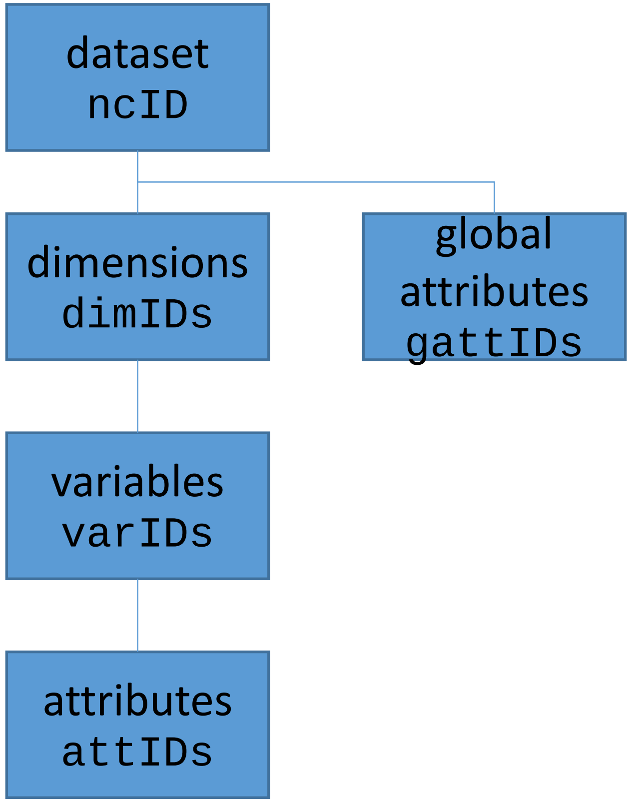
- Stores data in the form of multi-dimensional arrays;
- Underlying storage is abstracted away from user applications;
- Is portable across many different architectures, hence allows collaboration. It can be read by codes in other programming languages;
- Uses a highly optimised indexing system so data 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/m^2;
- Use the attributes to your advantage to describe your experiment and variables. This will help you share your data and avoid repeating the same simulation;
- Every NetCDF function should return `NF90_NOERR` constant.

Common Data Form Language (CDL) Example

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



Creating a NetCDF Dataset

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

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

Reading a NetCDF Dataset

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

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

Creating a NetCDF Dataset

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

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

```
int, 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 )
```

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.

NetCDF Commands (2)

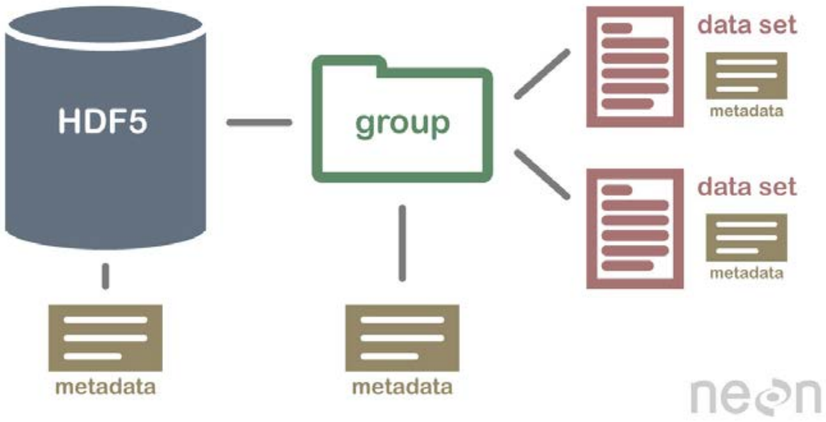
- `ncrename` - renames dimensions, variables and attributes in a NetCDF file;
- `ncra` - averages record variables in arbitrary number of input files;
- `ncwa` - averages variables in a single file over an arbitrary set of dimensions with options to specify scaling factors, masks and normalisations;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

HDF5 File Format

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

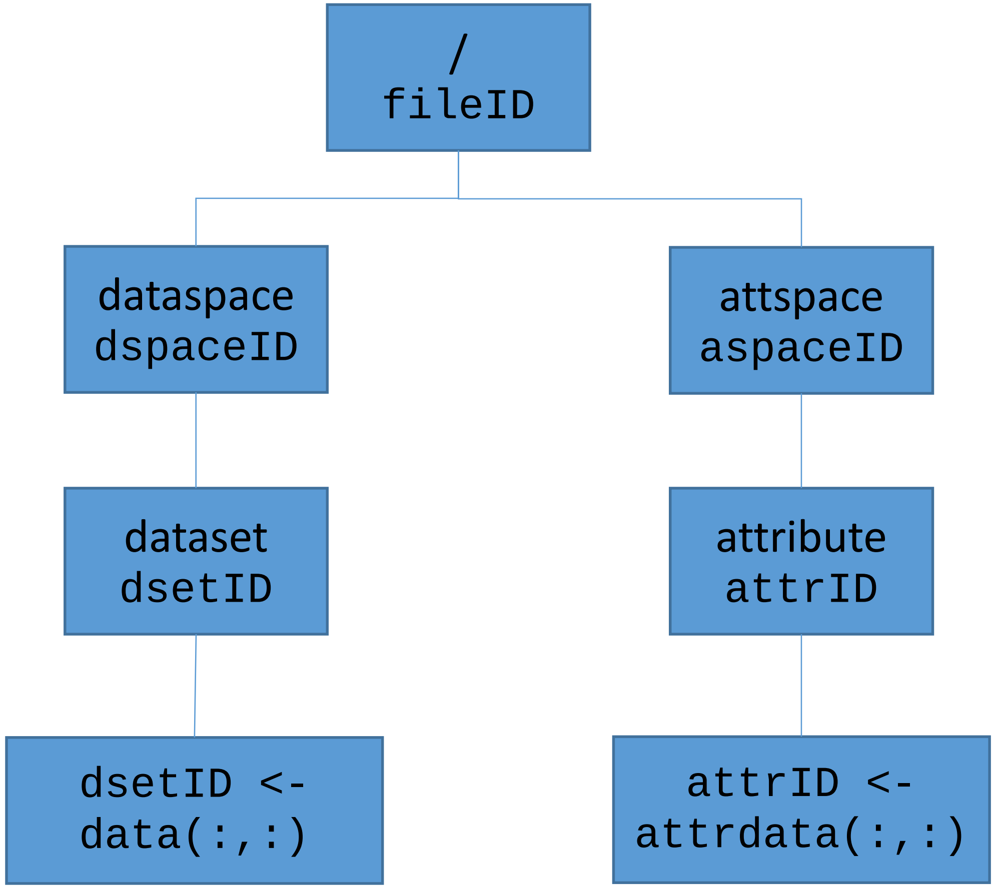
HDF5 Data Model

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



HDF5 Dataset Definition Language

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



HDF5 API Namespace

Prefix	Operates on
`H5A_*F( ... , ierr )`	Attributes
`H5D_*F( ... , ierr )`	Datasets
`H5E_*F( ... , ierr )`	Error reporting
`H5F_*F( ... , ierr )`	Files
`H5G_*F( ... , ierr )`	Groups
`H5I_*F( ... , ierr )`	Identifiers
`H5L_*F( ... , ierr )`	Links
`H5O_*F( ... , ierr )`	Objects
`H5P_*F( ... , ierr )`	Property lists
`H5R_*F( ... , ierr )`	References
`H5S_*F( ... , ierr )`	Dataspaces
`H5T_*F( ... , ierr )`	Datatypes
`H5Z_*F( ... , ierr )`	Filters

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

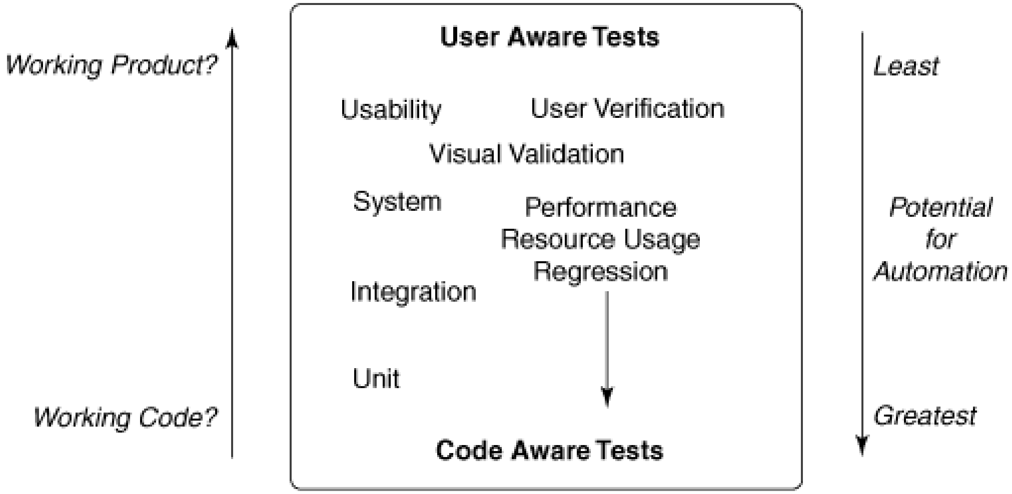
```
integer(kind = HID_T) :: file_id, dset_id, dspace_id, rank = 2
integer(kind = HSIZE_T), dimension(1:2) :: dims = [ 4, 6 ]

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

http://www.phdcomics.com

Testing Code (1)

- Testing is required to ensure the quality of the code [1]:



- Unit tests which verify particular paths through individual functions or subroutines;

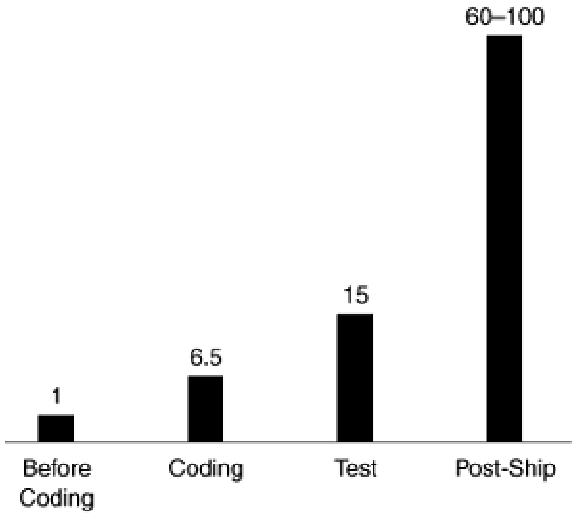
[1] “Sustainable Software Development: An Agile Perspective”, K. Tate. Addison-Wesley, 2005.

Testing Code (2)

- 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 $\epsilon_{\mathrm{model}}$, e.g. Navier-Stokes;
- Errors caused by domain discretisation ϵ_{disc};
- Truncation errors in numerical algorithms $\epsilon_{\mathrm{algorithm}}$, 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 $\epsilon_{\mathrm{float}}$, 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.

“Uncertainty Quantification: Theory, Implementation, and Applications”, R. Smith. SIAM, 2014.

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*;
- Mesh convergence testing - refine mesh and apply verification 1 and 2 above;

Testing in Computational Science (2)

- The infinity norm should be calculated and be within a certain tolerance ϵ;
- 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 Codes

- The compiler – prints a lot of diagnostic information as well as status of compilation;
- NAG Fortran compiler does extensive testing, including Fortran standards conformance 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;
- Eclipse Photran plugin does static analyses;
- CamFort for dimensional analysis on variables [3];
- pFUnit – unit testing framework for serial and parallel (OpenMP and MPI) codes.

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

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

Testing During Optimisation

- Results from sequential codes will vary between different optimisation flags which re-arrange calculations;
- Results from parallel codes will vary from run to run with fixed number of processes;
- Results from parallel codes will vary with different number of processes;
- This is due to process reductions being executed in different order due to the variability of which process reaches the reduction phase first/last;
- Codes that fail at larger process counts but work with small number of processes suffer from *scaling bugs*;
- A code whose performance suffers as a consequence of a change suffers from a *performance bug*.

Component Testing

- When developing a library/component, provide example codes that use your library for the purpose of testing;
- These tests will also allow the user to test a) the compiler b) MPI library c) the computational architecture;
- If using a library/component for your code, ensure you test the interaction of components;
- For n components, the maximum number of component interaction tests is given by $n(n - 1)/2$. This is known as *integration testing*.

Dimensional Analysis using CamFort

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.

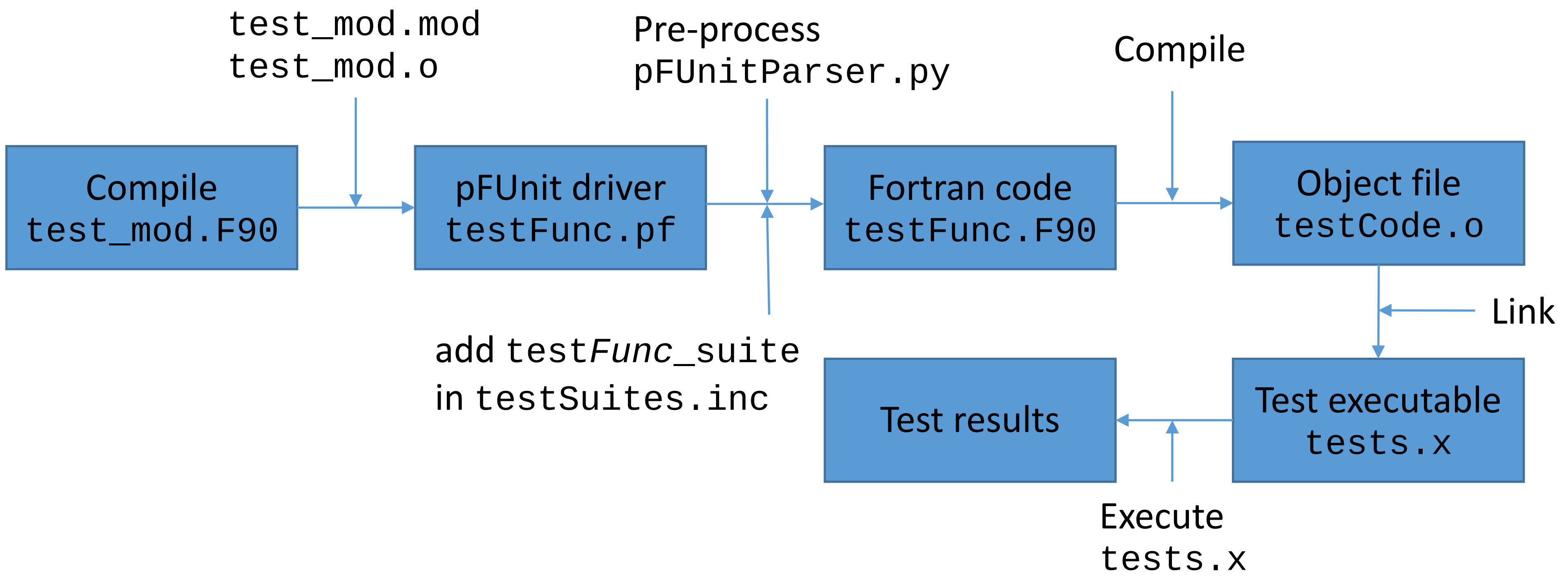
[1] http://artofunittesting.com/

FIRST Unit Testing

- **Fast** - many hundreds of tests per second;
- **Independent** - reason for failure is obvious;
- **Repeatable** - run repeatedly in any order and any time;
- **Self-validating** - no manual evaluation required, namely it is automated;
- **Timely** - written before the code, namely test driven development.

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 Factorial and Riemann

  integer :: result1

  real :: result2, tol = 0.00001

  result1 = Factorial( 2 )

  call Riemann( 2.0, 1.0, result2 )
  @assertEqual( 2 , result1 )

  @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]:

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

- Create the `tests.x` executable binary:

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

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

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

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.x`, 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

GCOVR - GUI Representation

- After tests are executed (`./tests.x`), use the Gcovr Python script [1] to summarise results `gcovr -r <dir>` where gcno and gcda files reside in `<dir>`;
- HTML files can be created using `gcovr -r <dir> --html \ -o output.html` which summarises code coverage;
- To get annotated code in HTML, type `gcovr -r <dir> --html \ --html-details -o output_details.html`

[1] http://gcovr.com/

Eclipse IDE - Photran (1)

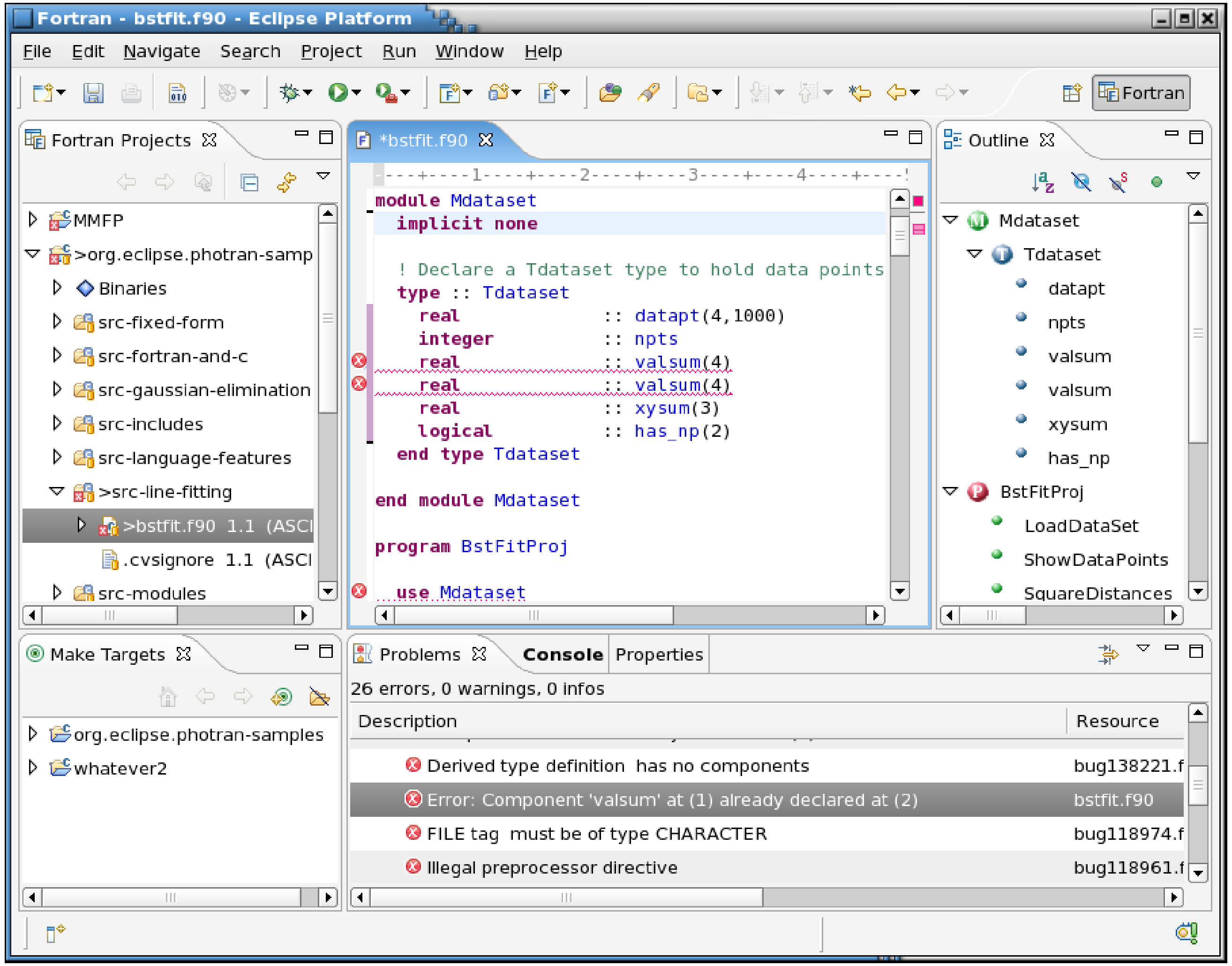
- Eclipse has a Photran plugin that has features for Fortran development. Version 9.1 supports Fortran 77 - 2008;
- Syntax highlighter;
- Outline view (expands user defined data types);
- Content assist - auto complete variable/function/subroutine names;
- Open declaration - finds the source of symbols in other modules;
- Declaration view - if you hover over a variable, function or subroutine, it shows its declaration;

http://www.eclipse.org/photran

Eclipse IDE - Photran (2)

- Fortran language based searching which allows regular expressions. Search contexts are subroutine, variable, function, module or program;
- Integration of GDB debugging and GCOV within Eclipse;
- Makefile based compilation;
- Limited refactoring features - from Fortran 77 to Fortran 90;
- Integration with GNU Fortran, Intel Fortran and IBM XLF compilers;
- Synchronises with HPC cluster file system so you can develop locally on your desktop.

Eclipse IDE - Photran (3)



Fortran Syntax Checkers for Linux Editors

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

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

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.

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)

- 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:

```
$FC -c -I/plplot/modules graph.F90
$FC graph.o -L/plplot/lib -lplplotf95d -lplplotf95cd \
            -o graph.exe
```

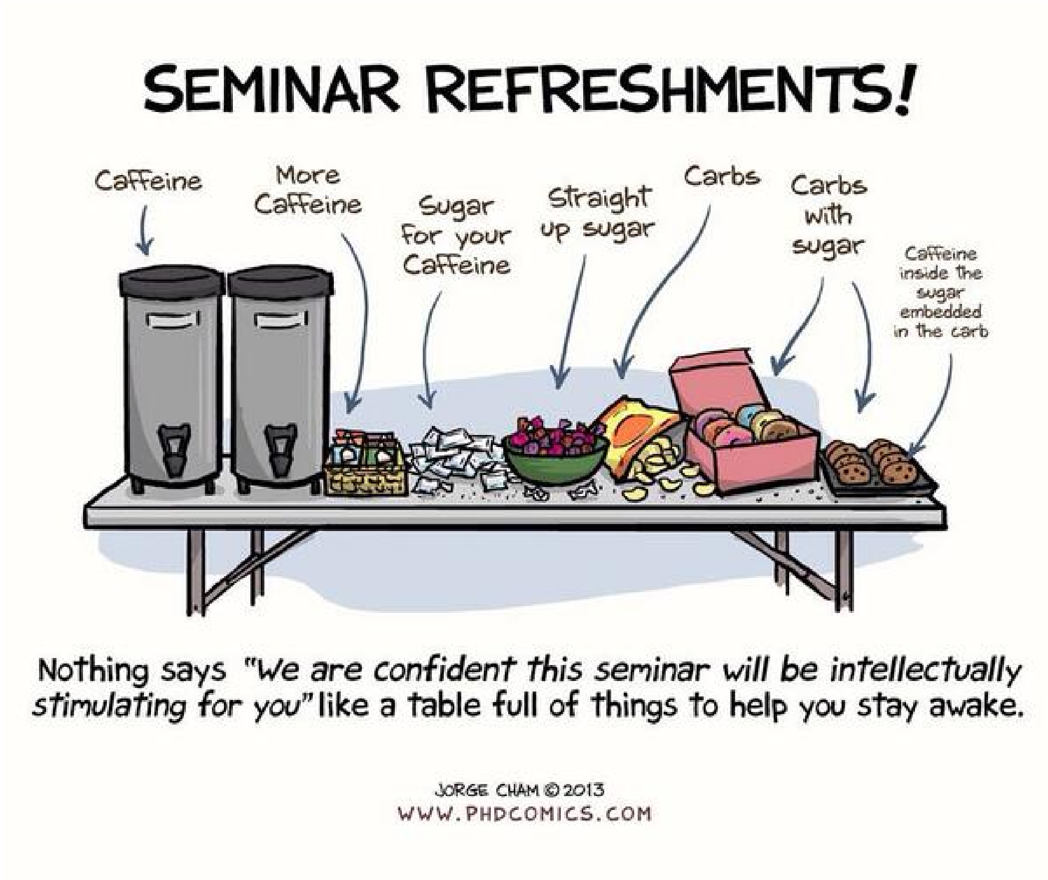
FFMPEG

- 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.

Lunch


SEMINAR REFRESHMENTS!
Caffeine
More Caffeine
Sugar for your Caffeine
Straight up sugar
Carbs
Carbs with sugar
Caffeine inside the sugar embedded in the carb
Nothing says "We are confident this seminar will be intellectually stimulating for you" like a table full of things to help you stay awake.
JORGE CHAM © 2013
WWW.PHDCOMICS.COM

Parallel Programming in Fortran

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;
- Fortran and C data type mapping is defined in the module;

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

Calling Fortran from C (1)

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

float reciprocal( float *x );

int main( ) {
 float x = 10.0f;

 printf(“1.0/%f = %f\n", x,
       reciprocal( &x ));
}
```

```fortran
! reciprocal_f.f90
function reciprocal_f( x ) result ( res ) &
bind(c, name = 'reciprocal')
  use iso_c_binding
  implicit none

  real(kind=C_FLOAT), intent(in) :: x
  real(kind=C_FLOAT) :: res

  res = 1.0 / x
end function reciprocal_f
```

Calling Fortran from C (2)

- Compile both files:

```
gfortran -c reciprocal_f.f90
gcc -c reciprocal_c.c
```

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

```
nm reciprocal_f.o
0000000000000000 T reciprocal
```

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

```
gcc reciprocal_c.o reciprocal_f.o -o reciprocal_c.exe
```

Calling C from Fortran (1)

```
! reciprocal_f.f90

program reciprocal_f
  use iso_c_binding
  interface
    real(kind=C_FLOAT) function reciprocal( x ) &
       bind(C, name = 'reciprocal_c')
       use iso_c_binding
       real(kind=c_float), value :: x
    end function reciprocal
  end interface

  real :: x

  x = 10.0
  print *, x, ' reciprocal = ', reciprocal( x )
end program reciprocal_f
```

```
/* reciprocal_c.c */

float reciprocal_c( float x )
{
  return 1.0f / x;
}
```

Calling C from Fortran (2)

- Compile both files:

```
gcc -c reciprocal_c.c
gfortran -c reciprocal_f.f90
```

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

```
nm reciprocal_c.o
0000000000000000 T reciprocal_c
```

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

```
gfortran reciprocal_f.o reciprocal_c.o -o \
reciprocal_f.exe
```

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(kind = DP), dimension(:), intent(in) :: array_f
     real(kind = DP), intent(out) :: result_f
     result_f = sum( array_f )
   end subroutine sumpy
   function fumpy( array_f ) result( result_f )
     real(kind = DP), dimension(:), intent(in) :: array_f
     real(kind = DP) :: 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):

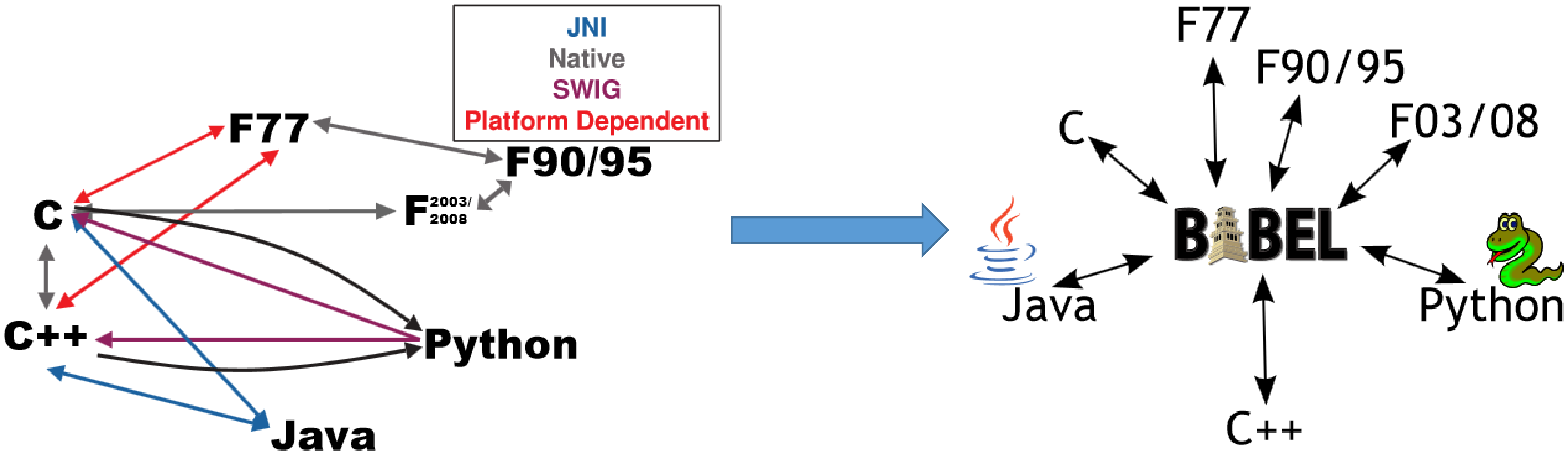
```
gfortran -c sums_mod.F90
gfortran -shared sums_mod.o -o sums_mod.so
```

- 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
```

Babel - Language Interoperability

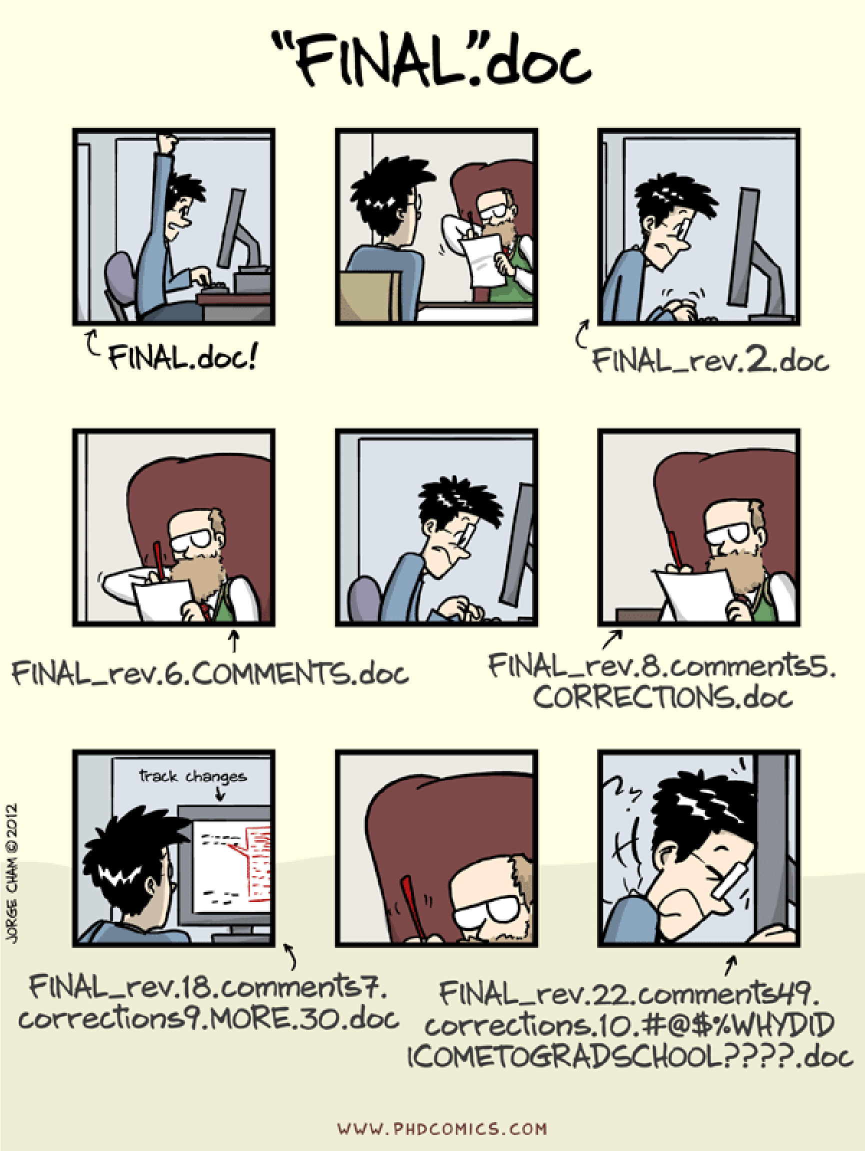
- Language interoperability can quickly get complex and simpler method is provided by Babel [1]:



[1] https://computation.llnl.gov/casc/components

Quick Break

Version Control

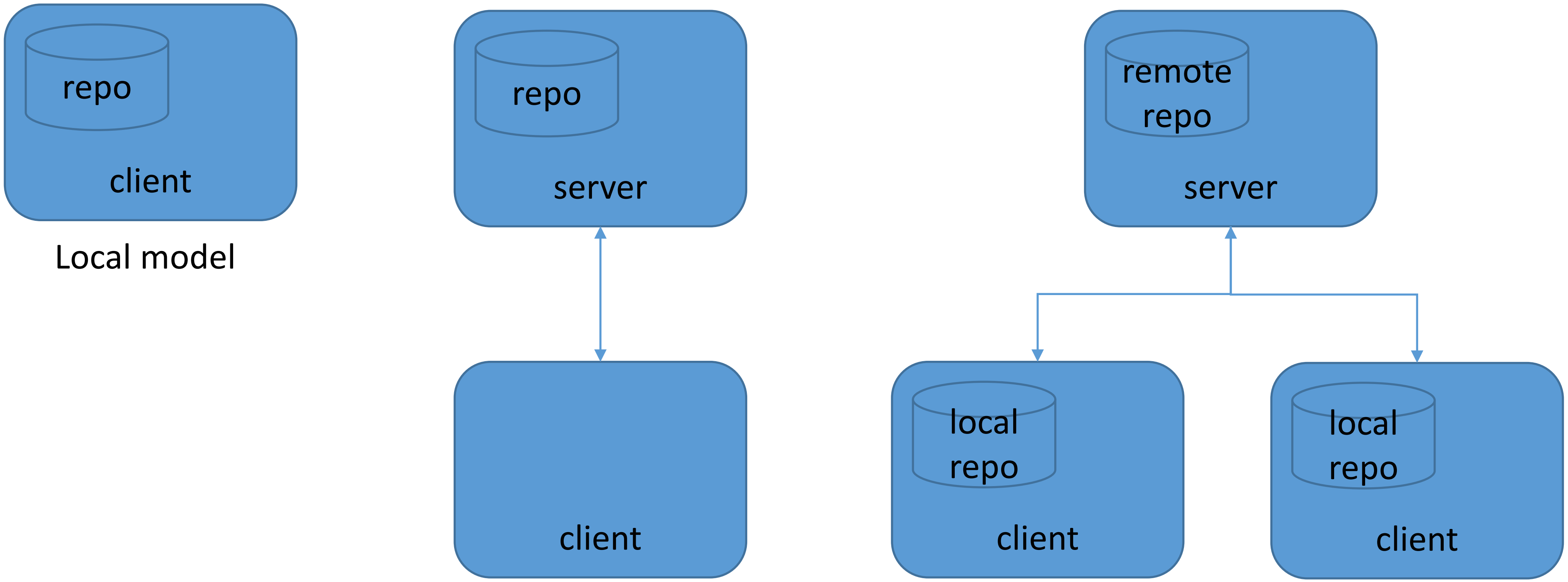


http://phdcomics.com

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*.

Types of Version Control Models



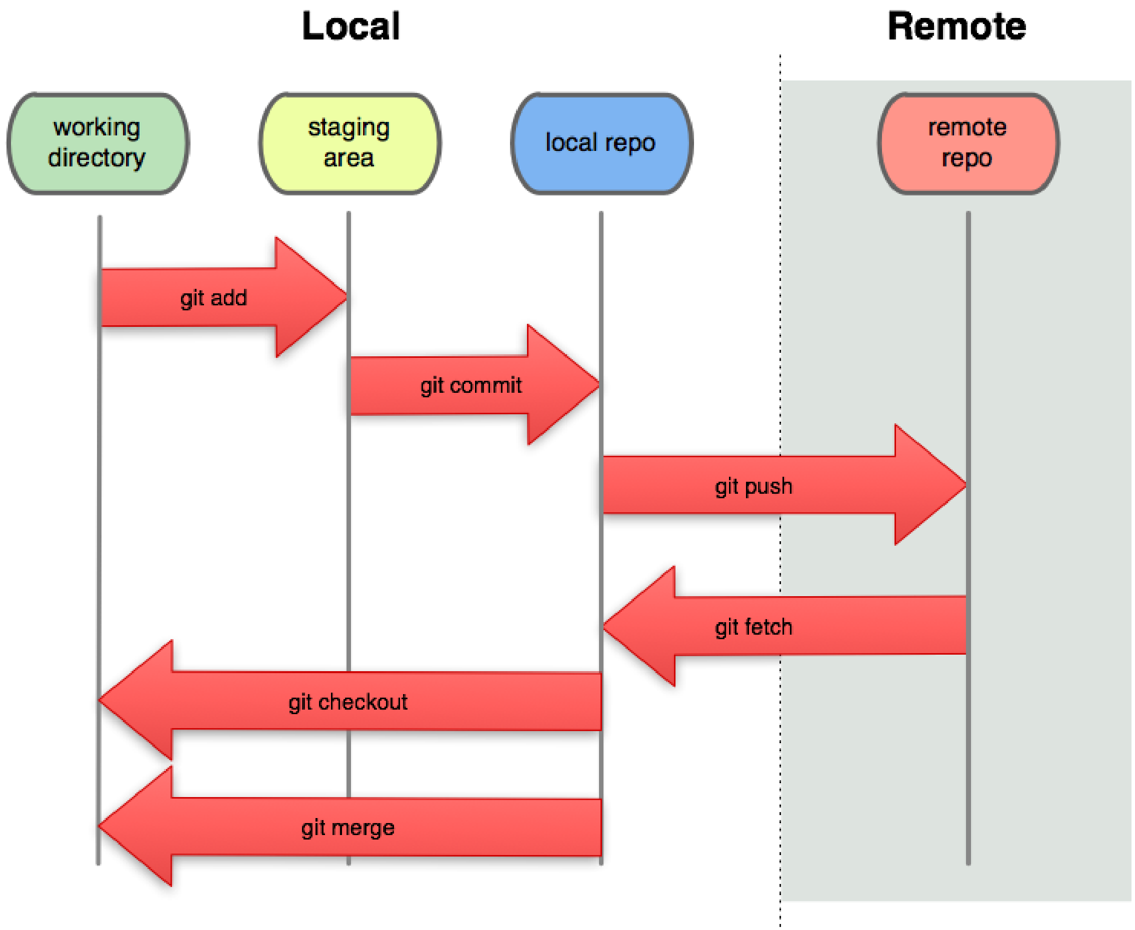
Local model

Client server model

Distributed model

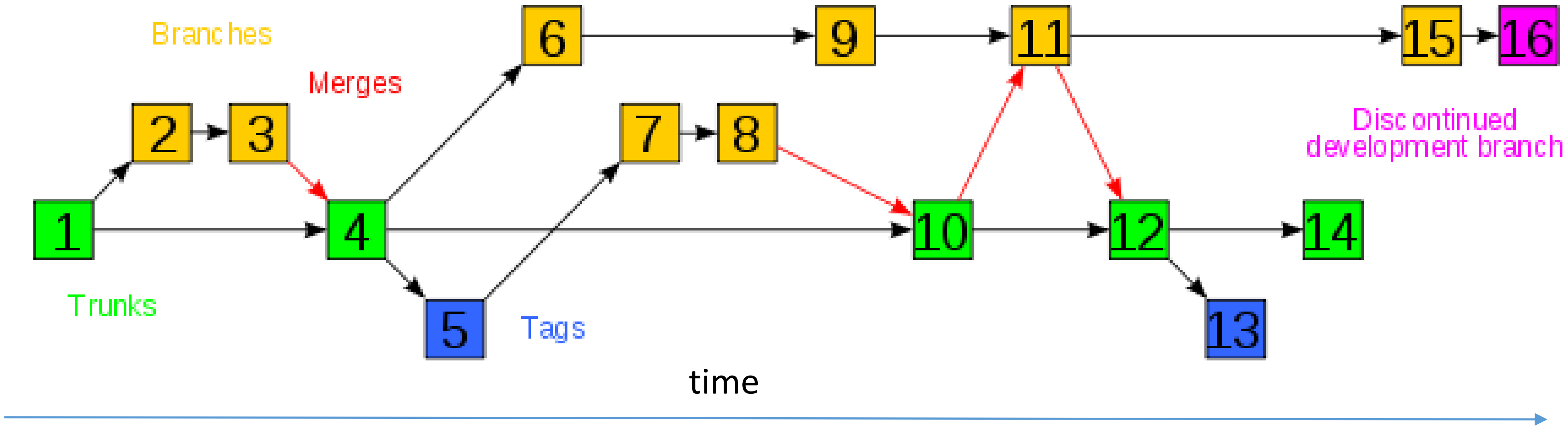
Git Version Control

- 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/doxygen
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
```

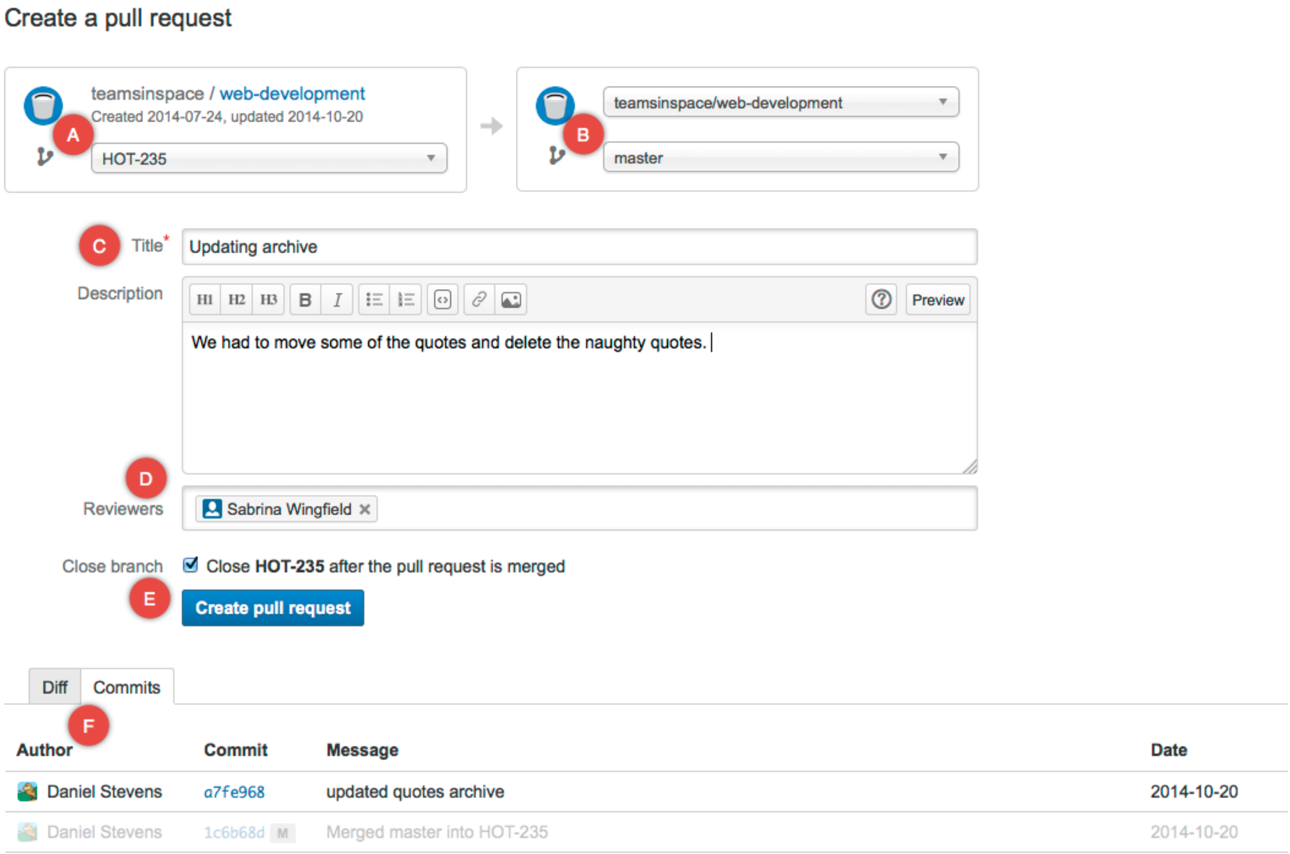
- Before making any changes to your local repository, always pull first;
- Push your changes after making your changes to local repository.

Collaboration Using Bitbucket

- Provides Git repository hosting and a ticketing system;
- Allows discussions between developers to aid collaboration;
- Provides fine grained access controls on branches;
- Stored on the could so your repository is backed up;
- Free unlimited public and private repositories for academic users;
- Provides a simple wiki and source code browsing. Access control is via SSH keys;
- Integration with your Continuous Integration system.

Merge Request - Bitbucket

- Merge request is a request to merge a branch into another branch;
- The request is reviewed which allows collaborative code review before merging;



Continuous Integration (CI)

- Testing tools such as GNU Make, pFUnit, gcov and gcovr can be integrated into a Continuous Integration (CI) system such as Jenkins [1] and Buildbot [2];
- CI systems allow automated building, testing and code coverage reporting using GCOVR;
- The build and test process is initiated when a user checks in their code using Git and reports can be emailed to developers;
- Ask your IT/HPC service to set up a CI system for your code.

[1] http://jenkins-ci.org/
[2] http://buildbot.net/

Software Release (1)

- Release notes should be provided [1];
- A set of tests to test the compiler and architecture;
- Installation instructions - should be clear and concise;
- Using the Linux tar command, package up your code and tar into a new directory `code-major.minor.patch` - this can be automated by your continuous integration tool;
- Get users to register with their email before downloading - keep a record of users to apply for funding to improve your code;

[1] https://en.wikipedia.org/wiki/Release_notes

Software Release (2)

- Select an appropriate licence for your code - see [2];
- *Create a compatibility matrix* - combination of compiler/MPI library that works and does not work for your code;
- If distributing binary executables, ensure all libraries are statically linked;
- For support, provide a group email address which has a number of developers as recipients of that group email address;
- This automatically produces support tickets for users who email that group email address;
- Communicate with your users through the wiki and mailing list.

[2] http://software.ac.uk/resources/guides/adopting-open-source-licence

Software Team Models

- Benevolent dictator for life - a single person makes all crucial project related decisions;
- Consensus driven - where the team makes decisions based on consensus and everyone is involved in the decision making process. This is the Agile model where everyone is trusted;
- Technical review board - a group of people which is a proper subset of the team make all crucial project related decisions;
- Which one should you choose? I would recommend the Agile method of consensus driven decision making. **Decisions take longer to make but are quicker to implement.**

Development Code of Conduct

- The code of conduct is a document that states the expected standards of behaviour from team members;
- They are good to have in the event of any conflicts between team members;
- Everyone should sign up to the code of conduct before joining the team;
- Examples of codes of conduct can be found at references [1] and [2] which are good starting points;
- It can also serve as a reminder to oneself on professional and ethical standards of behaviour particularly at difficult and challenging times.

[1] http://www.acm.org/about/se-code
[2] http://www.ubuntu.com/about/about-ubuntu/conduct

End of Day Two - Exercises 2

References (1)

- “The Art of Readable Code”, D. Boswell and T. Foucher. O’Reilly, 2011;
- “The Clean Coder”, R. Martin. Prentice Hall, 2011
- “Learning Agile”, A. Stellman and J. Greene. O’Reilly, 2014;
- “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;
- “Refactoring: Improving the Design of Existing Code”. M. Fowler, et al. Addison Wesley, 1999;
- “Managing Projects with GNU Make”, R. Mecklenburg. O'Reilly, 2004;
- “Software Engineering: The Current Practice”, V. Rajlich. CRC Press, 2011;
- “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.