THE UNIVERSITY of York

High Performance Computing - General Programming Techniques

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

Overview

- Software Lifecycle
- Program Design
- Coding Techniques & Defensive Coding
- Testing and Debugging
- Maintenance
- Software Reuse
- Patents and copyright

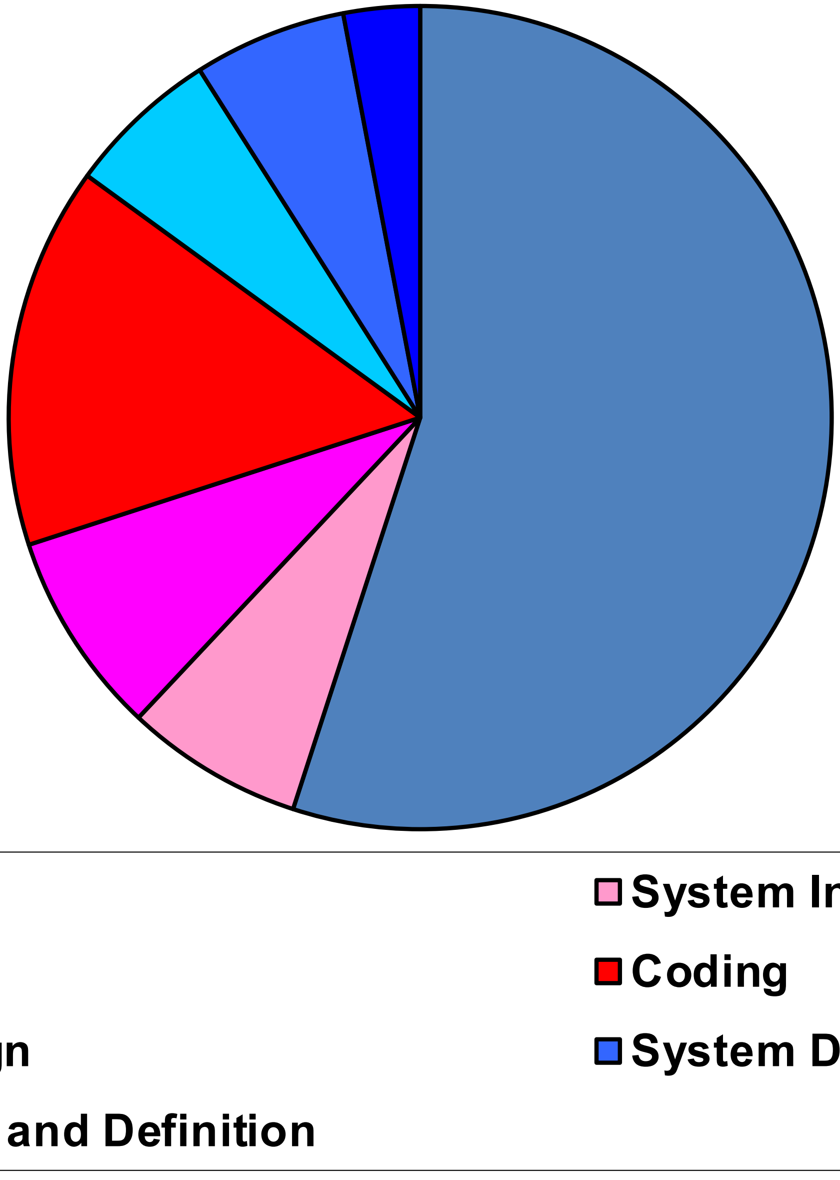
How to Start?

- Many people learn the basics of one or more programming languages
  - Far fewer people learn how to program
- Computer Science degrees emphasise programming theory etc, with whole modules on some of the things that will cover in this lecture
  - Less emphasis on fluency in any particular language
- What do we need for HPC?

Software Lifecycle

- IT is now of fundamental economic importance, with software costs being dominant
  - Many studies from 1980s onwards showed that software development and increasingly software maintenance costs were out of control
  - Hence a lot of interest in better understanding how to create quality software resulting in many competing theories and methodologies
  - Many well-documented examples of costly mistakes/disasters caused by bad software
- Often described in terms of a *software lifecycle*

Typical Proportion of Activities over a Software Product Lifecycle



Pre-Coding Phase

- Requirements
  - Clearly relevant for all HPC – need to have a clear understanding of the task
  - Need to sort out the physics of the problem, derive appropriate equations, algorithms, etc.
- Design
  - Having thought through the physics, it is NOT a good idea to sit down at a keyboard and start typing! Need to *design* program before start
  - Is this going to be a “one-off” program or will it be used more than once? By more than just you? Will it need to be adapted to similar tasks in the future?
  - Is there an existing code/library you can start from? Don’t re-invent the wheel unnecessarily!

Program Design

- Many different formal methodologies exist but most physicists ignorant of them!
- More than just draw a flowchart!
  - can be helpful for simple routines
  - rarely useful for complex problems
- Better to focus on the fundamentals of the physics
  - What are key quantities required and at what stage in the flow of the calculation? Input? Output? What operations are required on these objects?
  - Then think about how this might be implemented in any given language – F90 modules or C++ objects?
  - Consider "structured programming" and/or OOP

Data Structures (I)

The use of well-designed data structures can be a great help in coding:

• Conceptual simplification e.g. an atom type in an MD code – might want to *encapsulate* the different atomic positions & velocities, element types and masses

```
type :: atom
   real(kind=dp),dimension(3) :: r
   real(kind=dp),dimension(3) :: v
   real(kind=dp)              :: mass
   character(2)               :: atomic_symbol
end type atom
atom,dimension(N) :: myatoms
```

Instead of

```
real(kind=dp),dimension(3,N) :: r
real(kind=dp),dimension(3,N) :: v
real(kind=dp),dimension(N)   :: mass
character(2),dimension(N)    :: atomic_symbol
```

Data Structures (II)

Data hiding? Pass around a complex object and hide the internal representation, e.g. use

```
real(kind=dp)      :: Kinetic_energy = 0.0_dp
integer            :: i
do i = 1,N
    Kinetic_energy = Kinetic_energy + &
       calculateKE(myatom)
 end do
```

with

```
real(kind=dp) function calculateKE
implicit none
atom,intent(in) :: this_atom
calculateKE = 0.5_dp*this_atom%mass &
      *dot_product(this_atom%v,this_atom%v)
end function
```

so that main loop is blissfully unaware of the internals.

Data Structures (III)

Minimise scope for errors and localises coding changes, for instance if want to add some additional piece of information (e.g. number of electrons, isotope etc) or change the representation of some data item (e.g. from 2-character strings for element symbol to 20-character strings for element name

BUT beware – the code on the previous slides is VERY inefficient – poor memory access patterns and overheads!

May need to sacrifice clear data structures for speed.

Defensive Coding (I)

- Try to anticipate possible problems wherever possible and code to catch them
  - Bad style for your program to crash
  - Much better for it to politely inform the user of the problem in as much detail as possible and then shutdown gracefully (better for system stability too)
  - E.g. user input – never presume valid data
  - E.g. dividing by anything – could it be zero?
  - E.g. allocating/deallocating memory – use the built in 'stat=' facilities to catch errors
  - E.g. opening/closing files – use inquiry tests
  - All central to C / C++ code with try { } catch{ }
    - Not so central to Fortran
- Result will be longer but much more robust code

Defensive Coding (II)

- Valid inputs?
  - If code is expecting an integer, what if data = “fish”?
- Divide by zero?
  - What if denominator = 0?
- Memory allocation?
  - What if too big an array for available memory?
- Memory references?
  - What if array index is outside declared bounds of array?
- File I/O?
  - What if file does not exist? Or is write protected? Or disk full?
- …

Portability

- Code portability is important
- Why write portable code?
  - Just because you only need the answer on system X today does not mean you should limit yourself when system X becomes obsolete tomorrow!
  - Beware vendor/compiler specific language extensions and proprietary libraries
  - Check code works OK with different compilers on different platforms – very useful way of catching bugs!
  - Use standardised languages with appropriate compiler flags to enforce strict adherence to standards
  - Painful at first but worthwhile in the long run!

Efficiency

- Correctness is more important!
  - Often better to write a correct but inefficient version of an algorithm first
  - Then prove your code is correct
  - Then worry about efficiency later
- Will talk a *lot* more about efficiency of HPC codes in later lectures
  - Some problems have very bad scaling so it is always best to start with small version and then work up to actual problem want to solve

Coding

- Contrary to your expectations (and maybe your experience to date) coding generally takes a small fraction of whole activity!
  - Helps if proficient in language – i.e. aware of all keywords and built-in functionality
  - Helps more if understand basics of programming and the problem to be solved
  - Better still if have thought through requirements and design before you start …

    … and how to test/validate **small** chunks of code!

Testing

- Unit Testing
  - Think through at the design stage how the overall program can be built from smaller units
  - And how each of those units can be tested in isolation
  - And how to test the interfaces/interactions between these units in combination
- Be systematic!
  - Document the tests, inputs & outputs – Software QA!
  - Make sure invalid inputs are taken care of
  - Try to ensure all the code is covered in your testing! Analysis and testing tools can help here – more in later lectures.
  - Regression testing – make sure any new additions do not affect earlier functionality (need acceptance criteria)
- Modular approach to design, coding & testing

Validation

- What simple tests can you apply to validate a given unit? Or whole program?
  - Some simple theoretical limits (e.g. energy conservation in MD program)?
  - Specified inputs that give a known result? Also useful for checking later modifications!
  - E.g. numerical integration of TISE – check harmonic oscillator case for correct solution.
- Check robustness of code to compiler flags
  - Do all initial testing with maximum debugging & compiler support and NO optimisation
    - e.g. gfortran –g –Wall –fcheck=all –O0 etc
  - Only turn on compiler optimisation once code is correct and check it does not change anything significantly!

Debugging

- If testing shows a problem, how do you fix it?

Just as with testing, need to be systematic – use the scientific method!

1. Gather information and form a hypothesis
2. Test your hypothesis – understand what is wrong!
3. Iterate until proven hypothesis is found
4. Propose and test solution – not just a hack!
5. Iterate until solution is proven correct
6. Run regression test suite – has your fix broken something else that used to work?!
7. Extend test suite – add the inputs which triggered this failure to make sure it does not happen again!

Common Types of Bug

- Memory/Resource leaks
  - Make sure that everything allocated is always deallocated (once) when finished
- Logic Errors
  - Syntactically correct but code does not perform as expected. Impossible to catch with automated tools.
  - Needs rigorous testing of different sets of inputs to show up an odd/unexpected behaviour
- Coding Errors
  - E.g. parameter type and/or number mismatch, or exceeding valid input range of a routine, etc.

More Common Bugs …

- Memory Overruns
  - Beloved of hackers! Basically, trying to access a bit of memory that does not belong to you e.g. copying too long a string or going beyond an array bound.
  - Compiler flags can help catch this at runtime with some languages – e.g. compile with `-fcheck =bounds` for gfortran.
  - Common symptom – a strange numerical result or crash that goes away when you insert a print statement to try to see in more detail what is going on! Or when add an additional variable definition, etc.
  - This is a *Heisenbug* – an anti-fix – it masks the real problem and makes it appear to go away but in fact it makes the real problem even harder to find!

… and some more …

- Loop Errors
  - Infinite loops – make sure loop has a guaranteed exit strategy
  - Off-by-one loops – does your loop or array index go from 1 to N or 0 to N-1? Exit condition?
- Conditional Errors
  - Boolean logic mangled, e.g. testing for $x<0.0$ or $x\leq0.0$?
  - Beware nested 'if-blocks' – easy to get a missing 'else' clause. Always make sure that there is a catch-all clause.
  - Case statements better – add a 'case default'
- Pointer Errors
  - Memory style – beware uninitialised pointers (F90 as well as C-style languages although situation better in F95), pointers to deallocated blocks of memory, or pointers to wrong location

… and there's more …

- Integration Errors
  - i.e. when units pass tests OK but fail when put together
  - Usually due to inappropriate assumptions in one unit, e.g. will only work with data within certain ranges
- Storage Errors
  - What if your program wrote out an intermediate state of the calculation, in order to read it later to continue executing?
  - E.g. trying to create a file with an invalid filename, or file system becoming full, or file being locked by another application, or wrong access permissions, etc.
- Conversion Errors
  - SI vs. imperial units led to loss of NASA Mars Climate Orbiter in 1998. Converting 64-bit floating point variable to 16-bit signed integer led to loss of ESA Arianne 5 in 1996.
  - E.g. integer division, 12/24=0, but 12.0/24.0=0.5 …

… and finally …

- Hard-coded Lengths/Sizes
  - Magic numbers at random points in code makes changes hard to localise and code hard to maintain.
  - Much better to have assigned to variables or parameters with meaningful names and use name everywhere not value
- Versioning Bugs
  - Make file formats robust to future developments and build in versioning info – otherwise will get strange results when using old data in newer program version
- Inappropriate Reuse Bugs
  - Must always carefully unit test code, even when reusing old “trusted and reliable” code as it may be being used in a new way not originally planned & tested for!

Maintenance

- Software maintenance is a large cost, both financially and in terms of manpower – why?
- Different activities involved:
  - Bug fixes
  - Additional features
  - Performance improvements
  - Code tidying
- So where is the problem?

Causes of Maintenance Problems

- Poorly designed code – not designed for maintainability by original coder or others
- Poorly documented code – hard to understand what it was originally supposed to do, let alone after bug fixes & patches
- Hard to relate specific code to function
- Ripple effect – changes have a "knock-on" effect elsewhere and are not localised

Preventing Maintenance Problems

- Good Design
  - SP or OOP, with minimal global data, maximised use of data hiding, etc to localise effects of changes
  - In-built diagnostics that can be turned on/off at runtime
- Good Documentation
  - Original code *and* updates/fixes – else the documentation soon becomes obsolete
  - Can be in-line (comment statements, clear variable names and conventions, etc) – or Knuth's Literate Programming – or external document
  - And testing – sample inputs/outputs for regression testing, etc.
- Revision Control
  - Use hg/git etc. to manage changes

Software Reuse

- Reuse would appear to be a good thing
  - Idea championed in 1980s as a way of controlling software costs, increased productivity and reliability, reduced maintenance. OOP as a solution?
- Own codes? Not very often. Why not?
  - OOP not a magic bullet. Why? Requires more effort to design a general purpose, reusable class than a one-off.
  - OOP reuse with polymorphism can make bugs delocalised and hard to track.
  - OOP can result in deep class hierarchies that are very difficult to comprehend and hence maintain
- Use of libraries?
  - Reuse without modification can indeed result in less bugs, but if there are bugs it can take much longer to find them! Even worse if reused with modifications! Hence libraries good.

Patents and Copyright

- What about using someone else’s code without their express permission?
- Depends on licensing terms, copyright and patents.
  - Software patents very controversial – currently big business in USA and different rules in EU
  - Anyone can assert ownership of code by insertion of simple copyright statement into each file – “Creative Commons” project is seeking to improve this with a simple scheme of copyright agreements you can use “off-the-shelf”
  - Licensing is more popular – GPL and BSD are popular examples with important distinctions

Open Source / Open Data

- Open Source enables you to share your code and documentation with others
  - Can create a community of developers
  - Can freely reuse / modify / distribute code
- Open Data enables you to share your data with others – advocated by EPSRC etc
  - FAIR principles for digital assets
  - Findable, Accessible, Interoperable, Reusable
  - So put the data for your paper in a public archive, add metadata, DOI, etc

Further Reading

- “The Science of Debugging”, Matt Telles & Yuan Hsieh, Coriolis (2001)
- Software engineering https://en.wikipedia.org/wiki/Software_engineering
- Software Patents monitoring https://ffii.org
- Creative Commons https://creativecommons.org
- GNU licenses https://www.gnu.org/licenses/licenses.html
- BSD licenses https://docs.freebsd.org/en/articles/bsdl-gpl/
- Data management https://subjectguides.york.ac.uk/openresearch/data_code