Verification of Fortran Codes

Wadud Miah (wadud.miah@nag.co.uk)
Numerical Algorithms Group

http://www.nag.co.uk/content/fortran-modernization-workshop
Fortran Compilers

• Compilers seem to be either high performant or very good at error checking;
• There is a spectrum in between and compilers fall somewhere in between;
• Clearly the GNU and Intel compilers are mostly used, but how good are they at error checking?
Verification Features of Fortran Compilers

• Compiler vendors either focus their efforts on performance or good verification features (or maybe neither);

<table>
<thead>
<tr>
<th>Run-time Error</th>
<th>Absoft</th>
<th>g95</th>
<th>gfortran</th>
<th>Intel</th>
<th>Lahey</th>
<th>NAG</th>
<th>Pathscale</th>
<th>PGI</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage Passes (^1)</td>
<td>34%</td>
<td>45%</td>
<td>53%</td>
<td>53%</td>
<td>92%</td>
<td>91%</td>
<td>38%</td>
<td>28%</td>
<td>42%</td>
</tr>
<tr>
<td>TFFT execution time with diagnostic switches (^2)</td>
<td>10</td>
<td>16</td>
<td>6</td>
<td>12</td>
<td>445</td>
<td>60</td>
<td>19</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

• The two most commonly used compilers, namely Intel and GNU Fortran, are only able to detect 53% of defects in the benchmark suite;

• The NAG compiler is able to capture 91% of defects in the benchmark suite.

Usage of Verification Tools

• *Only 11 (7%) out of 155 Fortran developers are using verification tools;*

• Is there an over-reliance on compilers to detect defects? This certainly seems to be case;

• Advantage of verification tools is that they can detect bugs before a production run, namely during static analysis.
• There is anecdotal evidence to suggest that code verification is not considered important amongst Fortran programmers;
• This could obviously affect the quality of computational science codes.
Fortran Verification Workflow (1)

- Computational scientists obviously want correct code as well as fast code. What is the answer?
- Use both error checking and high performance compilers in tandem with automated verification tools;
- Static analysis tools still have limitations so the code still requires runtime checks with a good error checking compiler, e.g. NAG;
- Unit tests should be built with the NAG compiler with optimisations switched off. Use the following compiler flags with the NAG compiler:

  `nagfor -C=all -C=undefined -info -g -gline`
Fortran Verification Workflow (2)

- Integration tests should also be built with the NAG compiler with optimisations switched off;
- Once all tests have passed, then build with more performant compilers such as the Intel, Cray or IBM compilers.

Verification tools

Static analysis checks - CamFort, Forcheck, FPT

Rigorous standards checking and runtime checks

Intel, IBM or Cray compiler

High performance compilers

Fast and correct code
Fortran Verification Tools

• CamFort [1];
• FPT [2];
• Forcheck [3];
• NAG Fortran compiler [4];
• pFUnit is a unit testing framework [5];
• I will only very briefly discuss FPT, Forcheck and the NAG Fortran compiler.

Fortran Array Bug

• Spot the bug below:

```fortran
real, dimension(3) :: eng, aero
do i = 1, 3 ! 1 = port, 2 = centre, 3 = starboard
    aero = eng(i)
end do

! modern and correct version
aero( :) = eng( :)
```

• The FPT tool can detect the do loop bug.
Precision Bugs (1)

• The following code segments have bugs:

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

real(kind=REAL64) :: theta
real(kind=REAL32) :: x
x = 100.0_REAL64 * cos( theta )
```
Precision Bugs (2)

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

• Compilers are generally not good at spotting precision bugs;
• Compilers are not very good at detecting mixed precision bugs but the FPT and Forcheck tools can.
Forcheck Dummy Argument Checking

• Fortran code:

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

• Analysis output:

(file: arg_test.f90, line: 14)

B
**[610 E] optional dummy argument unconditionally used
Forcheck Dummy Argument Intent Checking

• Dummy arguments should always be scoped with the `intent` keyword;

• Command:
  forchk -intent arg_test.f90

• Analysis output:
  B
  **[870 I] dummy argument has no INTENT attribute
  (INTENT(IN) could be specified)
Forcheck Actual Argument Checking

• Fortran code:

    call foo( 1.0, b )

• Analysis output:

    7  call foo( 1.0, b )

    (file: arg_test.f90, line: 7)

    FOO, dummy argument no 1 (A)

    **[602 E] invalid modification: actual argument is constant or expression
Runtime Checking

• Static analysis checks are easy ways to detect obvious bugs but they are ultimately very conservative. When they say there is a bug, they are correct;

• Static analysis tools are limited in what they can achieve particularly for large multi-scale multi-physics code where there can be variables that are defined in complex IF statements;

• This requires runtime checks to ultimately check for potential bugs with a comprehensive error checking compiler such as the NAG Fortran compiler;

• The NAG Fortran compiler also prints helpful error messages to help locate sources of bugs instead of the dreaded “segmentation fault”.
NAG Compiler Optional Argument Detection

• Compile command (if Forcheck cannot detect this):
  nagfor -C=present arg_test.f90 -o arg_test.exe
• Fortran code:
  call foo( a )
  subroutine foo( a, b )
      real, intent(out) :: a
      real, intent(in), optional :: b
      a = b**2
  end subroutine foo
• Helpful runtime error message and not just segmentation fault:
  Runtime Error: arg_test.f90, line 14: Reference to OPTIONAL argument B which is not PRESENT
NAG Compiler Dangling Pointer Detection

• Build command:
nagfor -C=dangling p_check.f90 -o p_check.exe

• Fortran code:
real, dimension(:), allocatable, target :: vec
real, dimension(:), pointer :: vec_p

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

• Runtime output - NAG compiler is the only Fortran compiler that can check this:
Runtime Error: p_check.f90, line 12: Reference to dangling pointer VEC_P
Target was DEALLOCATED at line 10 of pointer_check.f90
NAG Compiler Undefined Variable Detection

• Compile command:
  nagfor -C=undefined undef_test.f90 -o undef_test.exe

• Fortran code:
  real, dimension(1:11) :: array
  array(1:10) = 1.0
  print *, array(1:11)

Runtime output:
  Runtime Error: undef_test.f90, line 7: Reference to undefined variable ARRAY(1:11)
  Program terminated by fatal error
NAG Compiler Procedure Argument Detection

• Compile command:
nagfor -C=call sub1.f90 -o sub1.exe

• Fortran code:
integer, parameter :: x = 12
call sub_test( x )
subroutine sub_test( x )
   integer :: x
   x = 10
end subroutine sub_test

• Runtime output:
Runtime Error: sub1.f90, line 13: Dummy argument X is associated with an expression - cannot assign
NAG Compiler Integer Overflow Detection

• Compile command:
nagfor -C=intovf ovf_test.f90 -o ovf_test.exe
• Fortran code:
  integer :: i, j, k
  
j = 12312312; k = 12312312
  i = 12312312 * j * k
• Runtime output:
  Runtime Error: ovf_test.f90, line 7: INTEGER(int32) overflow for 12312312 * 12312312
  Program terminated by fatal error
Conclusion

• More needs to be done to make code verification in computational science a mature practice just as it is in computer science;

• Develop a well-defined verification workflow and offer it as a service to the academic computational science community in the UK. *Verification as a service?*

• Promote verification tools and techniques at CDTs, HPC services, computational science community events and groups;

• Just teaching a programming language is wholly insufficient. Code developers need much more support;

• Every community code should openly publish the results of their unit tests and tool verification results to quantify the quality of their code.