# Verification of Fortran Codes

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

Numerical Algorithms Group

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

#### Verification Features of Fortran Compilers

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



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.

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

#### Usage of Verification Tools

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

Is there an over-reliance on

#### What Interests Fortran Programmers?



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.



#### **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.

[1] <u>https://github.com/camfort/camfort</u>
 [2] <u>http://www.simconglobal.com/</u>
 [3] <u>http://www.forcheck.nl/</u>
 [4] <u>https://www.nag.co.uk/nag-compiler</u>
 [5] <u>http://pfunit.sourceforge.net/</u>

# Fortran Array Bug

#### Spot the bug below:

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:

real(kind=REAL32) :: a, geom, v, g\_p

a = geom \* v \*\* (2/3) ! calculate surface area $<math>q_p = 6.70711E-52$ 

```
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

# Forcheck Dummy Argument Intent Checking

Dummy arguments should always be scoped with the intent keyword;

Command:

forchk -intent arg\_test.f90

Analysis output:

#### В



### 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

#### 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 Error: p\_check.f90, line 12: Reference to dangling pointer VEC\_P

Target was DEALLOCATEd at line 10 of pointer\_check.f90

#### NAG Compiler Undefined Variable Detection

Compile command:

nagfor -C=undefined undef\_test.f90 -o undef\_test.exe
Fortran code:

```
real, dimension(1:11) :: array
```

```
array(1:10) = 1.0
```

```
print *, array(1:11)
```

#### Runtime output:

```
Runtime Error: undef_test.f90, line 7: Reference to undefined variable ARRAY(1:11)
```

```
Program terminated by fatal error
```

#### NAG Compiler Procedure Argument Detection

Compile command:

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

Fortran code:

```
integer, parameter :: x = 12
```

```
call sub_test( x )
```

```
subroutine sub_test( x )
```

```
integer :: x
```

```
x = 10
```

end subroutine sub\_test

Runtime output:

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

#### NAG Compiler Integer Overflow Detection

**Compile command:** nagfor -C=intovf

#### 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?