Computer ‘Arithmetic’

What Every Computer Scientist Should Know About Floating-Point Arithmetic

http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Integer Arithmetic

- Fortran does this ok, just need to watch out for numbers (including intermediate results) too large/small to express. Operations then lead to wrapping.
- Sometimes 32 bit integers are not enough
- A sanity test for sufficient integer range: see int_sanity.f90

Integer Arithmetic

B=C=D=5000

E = B*D = 25000000 Right

A = E*C = 445948416 Wrong

print*, A/C 89189 But this is E!

```
ntotal = n*n*n
if (ntotal /= n*INT(n,8)*n) call panic(...)
```

B=C=D=5000

Floating point Exceptions

• Beside yielding approximate results, floating point operations can yield:
• Overflow, typically output as Inf - beyond the range of the fp numbers
• Underflow - closer to zero than representable, usually reset to zero. Watch for division by an underflow!!!
• Not mathematically permissible:log(0), sqrt(-1.)
Usually output as NaN (not a number)

See exceptions.f90

Floating point Arithmetic

- Floating point numbers are a finite subset of the rational numbers. They are bounded and have the same frequency per decade.

- In true arithmetic operations with rational numbers are also rational numbers.

- Beside the issue of boundedness, in FP arithmetic, an operation with FP numbers may only approximate an FP number - rounded or truncated.

FP Error Analysis

Let

$x_{fp} = x * (1+O(\varepsilon))$

$y_{fp} = y * (1+O(\varepsilon))$

Then

$(x_{fp} * y_{fp} - x * y)/(x * y) \sim O(\varepsilon)$

$\{x_{fp}+y_{fp} -(x+y)\}/(x+y) \sim O(\varepsilon)(|x|+|y|)/(x+y)$

In addition, if x and y are of opposite signs, x+y can be considerably smaller than either x or y, and the result is a larger relative error.

Key Floating Point Intrinsics

HUGE(x) - Largest non-infinite number of type x
TINY(x) - smallest positive number of type x
EPSILON(x) - smallest number E such that 1+E > 1
PRECISION(x) - decimal precision of type x
RANGE(x) - decimal exponent range

These intrinsic reveal the limits of floating point arithmetic for a particular Kind.
See fp_intrinsics.f90

FP Arithmetic - more consequences

Neither associative nor distributive:

(A+B)+C may not be A+(B+C) (ditto for *)

(A+B)-B may not be A (ditto for * and /)

A+A+A may not be 3.0*A

They do not have a multiplicative inverse: Not all A have a B = 1.0/A, such that A*B = 1.0

- Not continuous (for any of +, -, * or /):

  B > 0.0 may not mean A+B > A

  A > B and C > D may not mean A+C > B+D

  A > 0.0 may not mean 0.5*A > 0.0

See var.f90, sum_order.f90

A poorly-designed algorithm can be vulnerable to the vagaries of floating point arithmetic.

Worthwhile to use good third-party libraries.

Some Floating-point Best Practices

Do not compare two reals for equality:

```
IF (A == B) THEN
IF (ABS(A-B) < EPS) THEN
```

Protect against division by too small a number:

```
IF (ABS(c)> EPS) THEN
  a = b / c
ELSE
  a = d ! safety value for small denominator
ENDIF
```

When working with a large number of reals consider computing partial sums: example partialsums.f90

...and many more...

Code Optimization

- Three kinds of programming inefficiencies:
  - a computation-bound program
  - a memory-bound program
  - an IO-bound program

Efficient Computation

- AVOID DIVISION! It takes multiple clock cycles.

If you divide by the same number frequently, compute and save the reciprocal.

- Most CPUs can do an addition and multiplication simultaneously - write code the tends to pair them

a(i) = a(i) + b(i)*c(i)

Compiler Optimization

- Compilers have flags to perform optimization automatically, typically -O, -O2, -O3

- Compilers often have other flags to do faster but more approximate arithmetic.

Original loop:

```
do i = 1, ni
  a(i) = a(i) + b(i)*c(i)
enddo
```

Unrolled loop:

```
do i = 1, ni, 4
  a(i) = a(i) + b(i)*c(i)
  a(i+1) = a(i+1) + b(i+1)*c(i+1)
  a(i+2) = a(i+2) + b(i+2)*c(i+2)
  a(i+3) = a(i+3) + b(i+3)*c(i+3)
enddo
```

- Compiler optimization carries risk! Make sure your program gives (approximately) the same answer with and without optimization

Subroutine Bottlenecks

- Subroutine calls use lots of clock cycles which increase with the number of arguments. You want to maximize computation per subroutine call.

- Inlining is an effective way to optimize this. There are usually compiler tools to do this.

- Passing array subsections into subroutines may require a physical copy before any work is done.

Efficient Memory

- Memory efficiency is typically about managing cache use
- Avoid long strides - they tend to force the program out of cache more frequently

BAD:

```
DO i = 1, ni
  DO j = 1,nj
    a(i,j) = a(i,j) + b(i,j)*c(i,j)
  ENDDO
ENDDO
```

GOOD:

```
DO j = 1, nj
  DO i = 1,ni
    a(i,j) = a(i,j) + b(i,j)*c(i,j)
  ENDDO
ENDDO
```

IO Management

- IO bandwidth is much smaller than memory bandwidth.
- Access to local disks is faster than to remote filesystems.
- Unformatted IO avoids conversion of data and is faster than formatted IO.
- There are vendor specific techniques to overlap IO and computation - asynchrounous IO.

Where to Optimize? Profiling

- Profiling directs the programmer where to focus optimization efforts - work on the piece of code that consumes most of the time (wall-clock).

- Look for prof or gprof.

- Some platforms may have proprietary profilers, often need to use special compile flags.

- Other performance monitors tell you other metrics like flop-rate, cache usage, etc.

Debugging

- You've written your program, it compiles. Now you run it and it either crashes or gives bad output. You typically need to find out two things - where the error happens, and the variable values at that point.
- Types of errors:
  - Floating point exception - produces a number that is not a valid floating point number. To stop at an exception you need to enable floating point trapping
  - Segmentation fault (Sigsegv) or Bus error - often involve a memory error like a bad subscript or argument, often cryptic.

Simple Debugging

• Judicious use of write statements can help pinpoint where the program crashed and provide variable values.

• Every time you add write statements you must recompile. Also, an executable with write statements is not quite the one that blew up. Still, useful if you have a hunch where the problem is.

• Sanity checking - good program practice that tells you things are going wrong before they blow up.

```
if (speed > 0.0 .and. speed < 3.0e8) then
  …
else
  call panic('Speed error in my-function')
endif
```

Compiler Debug Tools

- Many compilers have options to provide extra error checking both during the compilation stage and during run-time. These do incur overhead and should be turned off when debugging is done.
- Array bounds checking: detects if an array subscript is out of bounds.
- Uninitialized variable initialization - all variables initialized with an FP exception. Together with FP trapping this lets you detect if you are using a variable before it is assigned a value.
- Actual option syntax is compiler-specific.

Compiler options for debugging

	Intel ifort	PGI pgf90	GNU gfortran
bounds-checking	-check bounds	-Mbounds	-fbounds-check
run=time detection of uninitialized variable	-auto -trapuv -check uninit		-finit-real=nan -finit-int=xxx -finit-logical=xxx
floating point trapping	-fpe 0	-Ktrap=divz,inv,ovf	-ffpe-trap=invalid,zero,overflow
debugger flags	-g	-g -O0	-g
generate stack trace	-traceback	-traceback	-fbacktrace
information	-diag-remark	-Minform-inform	

Where is the error?

- A program that terminates abnormally often generates a core file. The core file is a snapshot of the program contents at the time of crash. It includes a traceback (which program statement you are at) and the values of each variable.

- Some compilers generate the traceback apart from the core and write to standard output.

- There are debugger utilities to examine the core file. To use these you need to generate symbol tables, -g, in the object (.o) files, -c. Usually best to turn off all optimization as well.

Command line debuggers

- There are a host of similar command line debuggers that depend on the compiler: gdb (GNU), pgdbg (Portland group), idb (Intel).
- To examine a core file:
- gdb *executable_file corefile*; then at prompt type where.
- Examine values of variables:
- print i; print x(3,3)

Other useful debugger commands

- To see what the program is like before the crash

you can run it to a breakpoint: break *line_number*

gdb *executable_file*

break *line_number (no longer* stop)

r(un)

- execution commands:
  - r (run from beginning)
  - c (continue
  - s (step one line)
  - n (next line)
- Examine source code:
  - l (list 10 lines)
  - l *line_number*; l *filename.f90*
  - u (up one program level)
  - d (down one program level)

Graphical Debuggers

- Graphical interface debuggers greatly enhance debugging power. Examples are Totalview (widely available) and DDT (at NERSC).