Floating Point Arithmetic

* Floating point numbers are a subset of the rational numbers. However, they are bounded, and they have a finite density on the number line. Thus, your number may only be approximately represented in fp numbers, and results of operations involving floating point numbers may not reside among the floating point numbers.

Floating Point Approximations

* The result may lie between two fp numbers and will assume one of the two fp values
* The result may lie between zero and the smallest fp representation (in absolute value). This is **underflow**. Compilers typically set this to zero. Watch for division problems. Many compilers have options to otherwise flag underflows.

Floating Point Approximations

* The result may lie beyond the domain of fp numbers. This is **overflow**, typically output as **Inf**.
* The result may not be mathematically permissible - log(0.), sqrt(-1.), etc. This is **invalid operation**, typically output as **NaN** (not a number).
* This last three classes are known as **floating point exceptions**. Depending on the run environment, the program may terminate or continue when these are encountered.
* see **exceptions.f90**

FP Error Analysis

- x' = x*(1+d), y'=y*(1+e), d, e are relative error
- x'*y' = x*y*(1+e+d+...), e+d is the relative error, still small
- x'+y' = x+y + d*x + e*y, if x, y are of opposite signs, x+y can be considerably smaller than x or y, large relative error
- Algorithm design must account for vagaries of fp arithmetic, ill-conditioning.
- see var.f90

Code Optimization

- Three kinds of program inefficiencies:
  - a **computation-bound** program
  - a **memory-bound** program
  - an **i/o-bound** program

Efficient Computation

* AVOID DIVISION! - takes many clock cycles. If you will divide by the same number frequently, compute and save its inverse and multiply with it.
* most cpus can do an addition and a multiplication simulateously - write code statements that tend to pair them.

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

Compiler Optimization

- Compilers have flags to perform optimization, typically -O, -O3, -O4
- Compilers often have other flags to do quicker, but more approximate arithmetic.
- Compiler optimization techniques include loop-unrolling.
- Compiler optimization carries risk! Make sure your program gives the same answer optimized and unoptimized.

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
```

Subroutine Bottlenecks

* Subroutine calls use lots of clock cycles which increases with the number of arguments.
* Passing array subsections into subroutines requires a physical copy before any work is done.
* see sort_3a.f90, sort_3b.f90
* Inlining is an effective way to optimize this. There are usually compiler tools to do this.

Efficient Memory

- Memory efficiency is typically about managing cache use.
- Avoid long strides - 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
```

I/O Management

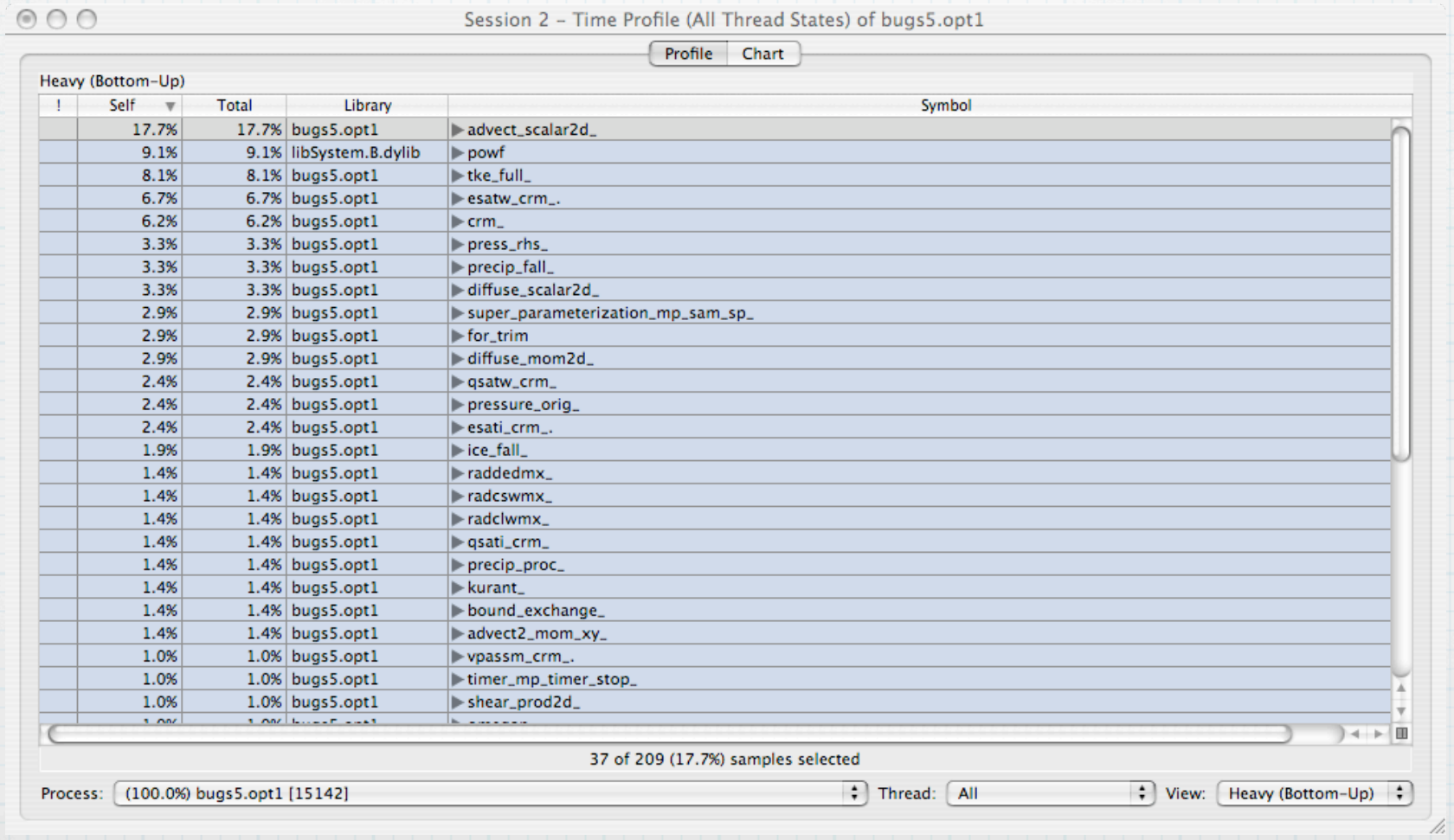
* I/O bandwidth is much smaller than memory. Local disks are faster than remote mounted filesystems.
* Unformatted I/O avoids conversion of data.
* Vendor specific techniques to overlap i/o and cpu usage - asynchronous i/o

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.
* On Mac OS X, use Shark, no special compilation necessary.
* Other platforms have proprietary profilers, often need to use special compile flags. Look for prof or gprof.
* Other performances monitors tell you the flop-rate, cache usage, other measures of performance.

Shark

* Launch /Developer/Applications/Performance Tools/Shark.app, start your program running, choose time profile, then start/stop Shark.



!	Self	Total	Library	Symbol
	17.7%	17.7%	bugs5.opt1	advect_scalar2d_
	9.1%	9.1%	libSystem.B.dylib	powf
	8.1%	8.1%	bugs5.opt1	tke_full_
	6.7%	6.7%	bugs5.opt1	esatw_crm_.
	6.2%	6.2%	bugs5.opt1	crm_
	3.3%	3.3%	bugs5.opt1	press_rhs_
	3.3%	3.3%	bugs5.opt1	precip_fall_
	3.3%	3.3%	bugs5.opt1	diffuse_scalar2d_
	2.9%	2.9%	bugs5.opt1	super_parameterization_mp_sam_sp_
	2.9%	2.9%	bugs5.opt1	for_trim
	2.9%	2.9%	bugs5.opt1	diffuse_mom2d_
	2.4%	2.4%	bugs5.opt1	qsatw_crm_
	2.4%	2.4%	bugs5.opt1	pressure_orig_
	2.4%	2.4%	bugs5.opt1	esati_crm_.
	1.9%	1.9%	bugs5.opt1	ice_fall_
	1.4%	1.4%	bugs5.opt1	raddedmx_
	1.4%	1.4%	bugs5.opt1	radcswmx_
	1.4%	1.4%	bugs5.opt1	radclwmx_
	1.4%	1.4%	bugs5.opt1	qsati_crm_
	1.4%	1.4%	bugs5.opt1	precip_proc_
	1.4%	1.4%	bugs5.opt1	kurant_
	1.4%	1.4%	bugs5.opt1	bound_exchange_
	1.4%	1.4%	bugs5.opt1	advect2_mom_xy_
	1.0%	1.0%	bugs5.opt1	vpassm_crm_.
	1.0%	1.0%	bugs5.opt1	timer_mp_timer_stop_
	1.0%	1.0%	bugs5.opt1	shear_prod2d_

Debugging

* You've written your program, it compiles. Now you run it and it either crashes, or gives you bad output - what do you do now? You want to find out two things - where the error happens, and the values of variables at that point.
* Types of errors:

  Floating point exception - divide by zero, other Nan operations. These need to be trapped, may need to enable **floating point trapping.**

  Bus error, Segmentation fault - often involve a memory error: bad subscript, bad argument passing, often cryptic.

Poor man's debugger - the write statement

* Judicious use of write statements can tell you where the program crashed and what values the variables had.
* Every time you add write statements you need to recompile. A slow, iterative process unless you already have a good hunch what has happened and you just want to confirm it.

Compiler debug tools

* Many compilers have options to provide extra checking both during the compilation stage and during run-time. These cause the program to incur overhead and should be turned off when the debugging is finished.
* Array bounds checking: detects if an array subscript is out of bounds
* uninitialized variable initialization - all variables are initialized with a floating point exception. This lets you detect if you are using a variable before it has been assigned a value.

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 platforms generate the traceback apart from the core.
* There are debugger utilities to examine the core file. To use these you need to compile your program with symbols **-g** flag. It is usually best to turn off compiler optimization as well.

Command line debuggers

* There are a host of similar command line debuggers that depend on the platform: **dbx, gdb, pgdbg, idb**
* To examine a core file: **gdb executable core**; then at prompt type **where**
* Examine values of variables: **print i; print x(3,3)**
* Usually, you find you need to back up in the program and examine variables prior to the crash point. Setting breakpoints lets you choose places to halt execution of the code: **stop line_number**

Other useful debugger commands

* Execution: **r** (run from beginning), **c** (continue), **s** (step one line), **n** (next line)
* Examine code: **l** (list 10 lines), **l line_number**, **list filename.f**
* **u** (up one program level), **d** (down one program level)

Graphical Debuggers

* The power of a debugger is enhance many-fold with a graphical user interface. Many sites support Totalview