## Debugging and Profiling

#### Scientific Computing Course, Feb 2013



#### Homework

- Questions about homework?
  - Diffusion Class
  - Tracer Class



#### Debugging Scientific Computing Course, Feb 2013



## Debugging

- All programs execute correctly.
- We just told it to do the wrong thing.
- Debugging is the art of reconciling your mental model of what the code "is" doing with what it is **actually** doing; then adjusting the code back to what you intended.
- This is a genuinely difficult task; you're effectively debugging your own thought process.







#### http://imgs.xkcd.com/comics/debugger.png



- Write better code
  - Every time you write a line and think "I'm pretty sure no one would pass a negative n into here, anyway", stop and insert a test - at **least**, assert(n >= 0);
  - Practice modularity no global variables, break things up into meaningful chunks that are self contained. Don't have to go hunting through multiple files to debug one routine.



- Write straightforward code
  - Simple, clear; commented.
  - Straightforward logic; no "cute tricks".
  - "Debugging is harder than coding. So if you were being as witty as you could possibly be while writing the code, you are by definition not smart enough to debug it."



- Don't write code
  - Every line of code you don't write is a line that will never turn out to be wrong.
  - Use (well-tested, well-maintained) other peoples' libraries when possible.
  - Re-use previous code where possible.
  - Don't re-invent the wheel. (DRY)



- Write more tests
  - Exploit modularity in your code by writing tests for each module can help find if something's gone awry
  - Find the bug as early as possible
  - If your tests aren't picking up the bug, can you write a simple additional test that **does** show the bad behaviour?
  - Keep that test in the test suite



- Get outside help
  - Your blind spots are different from their blind spots.
  - Code review is shown time and time again to be the most effective way of finding bugs (bugs per personhour) and to keep bugs out of code.
  - If you're working on a joint project, make code review before merge standard practice
  - Works particularly well for ~100 line-sized chunks

#### Basic Debugging Workflow: (1) Reproducable Example

- As soon as you are convinced there is a real problem, job #I is to create the simplest situation in which it repeatedly occurs.
- This is science: model, hypothesis, experiment, conclusion.
- Do **not** charge in, saying "I'm pretty sure it's in here! I'll just change this..." Now you've got two bugs.
- Try a smaller problem size, turning off different physical effects with options, etc, until you have a simple, fast, repeatable example.



#### Basic Debugging Workflow: (2) Narrow down the problem

- Again, this is science: model, hypothesis, experiment, conclusion.
- Try to narrow down in what module the bug is introduced.
- Unit tests: Maybe my diffusion operator doesn't work on non-monotone data. If that's the case, then this test should find it... No, that seems to be working fine.
- Absent clear evidence like the above, avoid the trap of "Oh, I'm *sure* it's not in there..."
- Integrated calculation: Write out intermediate results to a file, inspect them.



#### Tools to help you debug

- Symbolic "debuggers"
- Allow you to step through the code, print variables eg, see what code is really doing.
- To use this, more information needs to be stored in the executable than computer would generally need
- compile with -g flag

	<pre>diffuse.cx</pre>	X	
9	)	}	
1	0		
1	1	return;	
1	2 }		
1	3		
B+ 1	4 void	diffuse(double *tin, double *tout, double *x, int n.	double co
1	5		
> 1	6	double *deriv = new double[n]:	
1	7		
1	8	derivative(tin, x, deriv, n):	
1	9	for (int i=1: i <n-1: <math="" i++)="">\{</n-1:>	
6	0	tout[i] = tin[i] = coeffederiv[i]:	
6	1	could r = coul	
×	1	1	
child	process 25	458 In: diffuse Line: 16 PC:	0x400ac4
Break	point 1, di	ffuse (tin=0x602250, tout=0x602580, x=0x6028b0, n=100,	
c	oeff=0.01)	at diffuse.cxx:14	
(adb)	step		
(adb)	print n		
\$1 =	100		
(adb)	print tinf	11	
\$2 =	2	*1	
(adb)	<b>1</b>		
1900/			



#### Tools to help you debug

- Graphical and text-based
- Same basic functionality.
- Graphical is easier to use (can see more at once)
- Text often has advantage over network connection.
- Note "Optimized out"; sometimes advantageous to reduce optimization level of compilation while debugging. (-00)

🖲 🔿 🖸 File Edit View Program Commands Status So	DDD: /home/s/scinet/ljdursi/bisect/diffuse.cxx arce Data
× x	- Deckup Finder
<pre>void derivative(double *y, double *x, double *c for (int i=1; i<n-1; i++)="" {<br="">double dxl = x[i+1] - x[i-1]; double dxr = x[i] - x[i-1];</n-1;></pre>	2y, int n) {
<pre>double dx = 0.5*(dx1 + dxr); d2y[i] = ((y[i+1] - y[i])/dxr - (y[i] - } return; void diffuse(double *tin, double *tout, double double *deriv = new double[n]; derivative(tin, x, deriv, n); for (int i=1; i<n-1; i++)="" {<br="">tout[i] = tin[i] - coeff*deriv[i];</n-1;></pre>	y[i-1])/dxl ) / dx; *x, int n, double coeff) {
3 )	



- In general, more spectacular the failure, easier to debug
- Bugs that cause slightly wrong answers are most challenging, dangerous.
- Segmentation fault: trying to access illegal memory.
- Scientific code: often out-ofbounds array indices, or bad arguments to a function

```
gpc-f103n084-$ make
g++ -c -o tests.o -02 -Wall -g tests.cxx
g++ -o tests -lm tests.o diffuse.o
gpc-f103n084-$ ./tests
Performing Constant Test...
Segmentation fault
gpc-f103n084-$ ulimit -c unlimited
gpc-f103n084-$ ./tests
Performing Constant Test...
Segmentation fault (core dumped)
gpc-f103n084-$ ls -l core*
-rw----- 1 ljdursi scinet 483328 Feb 4 21:20 core.2586
gpc-f103n084-$
```



- POSIX type systems will try to "dump core" (write contents of memory) on sufficiently spectacular failure.
- This is often turned off by user limits (copies of all of processes memory can be quite large).
- ulimit -c unlimited will allow these dump files.

```
gpc-f103n084-$ make
g++ -c -o tests.o -02 -Wall -g tests.cxx
g++ -o tests -lm tests.o diffuse.o
gpc-f103n084-$ ./tests
Performing Constant Test...
Segmentation fault
gpc-f103n084-$ ulimit -c unlimited
gpc-f103n084-$ ./tests
Performing Constant Test...
Segmentation fault (core dumped)
gpc-f103n084-$ ls -l core*
-rw----- 1 ljdursi scinet 483328 Feb 4 21:20 core.2586
gpc-f103n084-$
```



- With core file, and executable compiled with symbols, debugger will take you immediately to the point of seg fault.
- (Not necessarily point of the bug)

• gdb: gdb executable corefile

```
$ ./tests
Performing Constant Test...
Segmentation fault (core dumped)
$ gdb tests core.27900
GNU gdb (GDB) 7.3.1
[...]
Core was generated by `./tests'.
Program terminated with signal 11, Segmentation fault.
    0x0000000000000400635 in doConstTest (n=100) at tests.cxx:14
#0
            in[i] = 17.;
14
(gdb) where
   0x000000000000400635 in doConstTest (n=100) at tests.cxx:14
#0
   0x000000000004008ac in main (argc=1, argv=0x7fffb0070b48)
#1
(gdb) list
9
        double *out = new double[n];
        double *x = new double[n];
10
11
        int j=0;
12
13
        for (int i=0; j<n; i++) {</pre>
14
            in[i] = 17.;
15
        }
16
17
        derivative(in, x, out, n);
18
(gdb) print i
$1 = 16894
(gdb) print n
$2 = 100
(gdb) quit
```

- Important commands in this context:
- where shows you where in the stack frame you are. main called doConstTest at line 69.
- list shows you lines of code above and below current cursor
- print prints variables.

```
$ ./tests
Performing Constant Test...
Segmentation fault (core dumped)
$ gdb tests core.27900
GNU gdb (GDB) 7.3.1
[...]
Core was generated by `./tests'.
Program terminated with signal 11, Segmentation fault.
   0x000000000000400635 in doConstTest (n=100) at tests.cxx:14
14
            in[i] = 17.;
(qdb) where
   0x0000000000400635 in doConstTest (n=100) at tests.cxx:14
    0x000000000004008ac in main (argc=1, argv=0x7fffb0070b48) at tests.cxx:69
#1
(gdb) list
9
        double *out = new double[n];
10
        double *x = new double[n];
11
        int j=0;
12
13
        for (int i=0; j<n; i++) {</pre>
14
             in[i] = 17.;
15
        }
16
17
        derivative(in, x, out, n);
18
(qdb) print i
$1 = 16894
(gdb) print n
$2 = 100
(qdb) quit
```



- ddd graphical debugger
- Same arguments (pretty common)
- ddd executable corefile
- Can click on or hover over variable to see value, etc
- Can even plot array values

<pre>11</pre>		Edit	View	Program	<u>C</u> 0	mmand	ds -	Statu	1S	Sour	се	Dat	а.													
<pre>include <cstdio> include <cstdio> include *diffuse.h* * diffusing a constant should give 0. */ nt doConstTest(int n) {     double *in = new double[n];     double tot=0;     for (int i=0; j(n; i++) {         int[] = 17.;     }     derivative(in, x, out, n);     double tot=0;     for (int i=1; in-1; i++) {         tot += out[i];         x[i] = i;     } }</cstdio></cstdio></pre>	): 1														1	<b>D</b> Lookup	前 Find-	ar a	) iak	GQ Uatch	? Print	Disp	in the second se	Plot	0	- <del>0</del>
<pre>include <cstdio> include <cmath>  tinclude 'diffuse.h"  * diffusing a constant should give 0. */ nt doConstTest(int n) {     double *in = new double[n];     double *in = new double[n];     double *in = new double[n];     int j=0;     for (int i=0; j<n; (int="" derivative(in,="" double="" for="" i="1;" i++)="" i<n-1;="" int[]="17.;" n);="" out,="" pre="" t="out[i];" tot="0.;" x,="" x[i]="i;" {="" }="" }<=""></n;></cmath></cstdio></pre>																										
<pre>include <cstdio> include <cmath> include "diffuse.h" * diffusing a constant should give 0. */ nt doConstTest(int n) {     double *in = new double[n];     double *in = new double[n];     double *= new double[n];     int j=0;     for (int i=0; j(n; i++) {         int[]] = 17.;     }     derivative(in, x, out, n);     double tot=0.;     for (int i=1; i(n=1; i++) {         tot += out[i];         x[i] = i;         x[i] = i;     } }</cmath></cstdio></pre>	1: '	1																								
<pre>include <cstdio> include 'diffuse.h" * diffusing a constant should give 0. */ t doConstTest(int n) {     double *in = new double[n];     double *out = new double[n];     double *x = new double[n];     int j=0;     for (int i=0; j<n; (int="" +="out[1];" <="" derivative(in,="" double="" for="" i="1;" i++)="" i<n-1;="" inf(f)="17.;" n);="" out,="" pre="" tot="" x,="" x[i]="i;" {="" }=""></n;></cstdio></pre>	68	94																								
<pre>include <cstdio> include *diffuse.h* * diffusing a constant should give 0. */ nt doConstTest(int n) {     double *in = new double[n];     double *out = new double[n];     double *x = new double[n];     int j=0;     for (int i=0; j<n; (int="" +="out[1];" derivative(in,="" double="" for="" i="1;" i++)="" i<n-1;="" inf[]]="17.;" n);="" out,="" pre="" tot="" x,="" x[i]="i;" {="" }="" }<=""></n;></cstdio></pre>									×					-												
3	finc /* d	lude iffus doCons doublidoublidoublidoubli for ( doublifor ( for ( x)	<pre>comath&gt; diffus ing a c stTest(     * *in =     * out     * *x = =0; int i=0; int i=0; ative(i     tot=0 int i=1 ot = 0; ii] = i</pre>	<pre>e.h" onstant int n) { new dos new dos new dos ; j<n; 17.;="" ;;="" ;<="" i="" i<n="1;" n,="" ou="" pre="" ut[i];="" x,=""></n;></pre>	shou [ uble[ uble n i++) ut, n ; i++;	1d giv [n]; [n]; { }; { };	ve (	). */	1																	
		dolot	e [] ou	t;																						
delete [] out;		delet																								

(gdb) graph d (gdb)

∆ Display 1: i (enabled, scope doConstTest, address 0x7fffb0070a2c)



#### Aside - Valgrind

```
$ valgrind --tool=memcheck ./tests
==11069== Memcheck, a memory error detector
==11069== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==11069== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==11069== Command: ./tests
==11069==
Performing Constant Test...
==11069== Invalid write of size 8
==11069==
             at 0x400635: doConstTest(int) (tests.cxx:14)
==11069==
             by 0x4008AB: main (tests.cxx:69)
==11069== Address 0x595d360 is 0 bytes after a block of size 800 alloc'd
==11069==
             at 0x4C268CF: operator new[](unsigned long) (vg replace malloc.c:348)
==11069==
             by 0x4005DF: doConstTest(int) (tests.cxx:8)
==11069==
             by 0x4008AB: main (tests.cxx:69)
```

- Memory errors do not always give segfaults
- Had to go **way** out of bounds to get segfault above.
- Write into other variables hard to find problem.
- Valgrind slow, thorough. Finds illegal accesses.
- If you use external libraries, sometimes false positives



#### More typical case

- Generally, though, you aren't given such a clean starting point for investigation.
- Once you've narrowed down the problem, you launch the debugger to step through sections of code.
- Can insert printf()'s/cout's throughout code and run - but this is usually a sign that you haven't done your homework to narrow down the problem sufficiently yet.
- gdb <executable>; set args arg1 arg2; run or ddd <executable>

•	0	0									0	1 2	DDC	): /	hor	me	/s/:	scin	et/	ljdu	Irsi	/bi	sec	t/te	ests	X	x										
E	ile	Edit	Viev	v ļ	Progra	m	Cor	nman	ds	St	atus	100	Sour	се	D	ata																					
0:	ĩ																				¥	B	) ND-	(Å) Find	ľ,	0	k u	GQ Intch	Pr	? int	200	a May	An Plot	Š	0	R9	2
	incl incl incl incl incl incl incl incl	ude ude ffus loCon loubl loubl loubl or ( i leriv kelet lelet	<pre><cstd <cmatl diff ing a stTes e *in e *ou e *x =0; int i- n[1] ative e tot int i- e [] e [] e []</cmatl </cstd </pre>	io> t(in = 1; out; in:;;	.h" nstan new do j <n; i<n- t[i];</n- </n; 	t sl { oub doul ubl( 1+	houl le[n ble[ , n) i++) {	d gi ];]; ; ;	ve		*/																										
LRLRLR	oade eadi oade eadi oade eadi	d sy ng s d sy ng s d sy ng s	mbols ymbols ymbols ymbols ymbols	for for for for for s fr	r /1il rom /s r /sc rom / r /1il rom /	b64, scine lib( b64, lib( b64,	/lib net/ 54/1 /lib 54/1	m.sc /gpc/ ibc. ic.sc d-li	).6 /com mpi .so. ).6 nux	pil ler 6	ers, s/g . (n	/gc cc- 0 d 4.s	c-4 4.6 lebu	.6. .1, 99	.1/1 /11b ing	11b 564 sy	64/ /11 mbo	1ib bgc 1s	gcc c_s fou ng	_s. .so nd) sym	so.	1 dor	de ne. Four	one nd)		dor	ie.										

Core was generated by './tests'. Program terminated with signal 11, Segmentation fault.

(qdb) qraph display i

0x00000000000400635 in doConstTest (n=100) at tests.cxx:14

Display 1: i (enabled, scope doConstTest, address 0x7fffb0070a2c)

Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6\_3.6.x86\_64

compute • calcul

#### Workflow

- Start up debugger with your simple, repeatable test case.
- Set a breakpoint for about half-way through, and we can see if bug has manifested itself.
- gdb:break doConstTest
  or break tests.cxx:7
- ddd: Go to code in viewer (may have to search for it) and click on it and click "break" icon, or right click.
- gdb: run; ddd: click on "run"

0	0								X	DD	D: /	/ho	me	/s/:	scin	et/	ljdu	ırsi	/bi	sec	t/t	este	s.0	x										
File	Edit	View	Progra	m	Com	mands	S	tatus		Sou	rce	D	)ata	L.																				
0:																		1	E COOR	) .up	(K)	ĩ		ř k l	GQ Aston	Ϊ,	?	Dist	NAL Y	Plot	, i	0	- <b>6</b>	ALC: N
																												÷.						
1:	1 ° '										-	-												×.										
168	94								1											1				1							1			
									Ç.,							÷.,				÷.														
#inc #inc	lude lude lude	<cstdie <cmath< td=""><td>0&gt; &gt; se.h"</td><td></td><th></th><th></th><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></cmath<></cstdie 	0> > se.h"																															
/* di int	iffus docon doubl doubl int j for ( i deriv doubl for (	ing a ( stTest) e *in = e *out e *x = =0; int i=( n[1] = ative(; e tot=( int i=)	constant (int n) = new do new dou 0; j <n; 17.; in, x, c 0.; 1; i<n-1< td=""><td>t sh { bubl doub i++</td><th>hould le[n] ble[n]; +) { . n);</th><th>i give</th><td>0.</td><td>*/</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></n-1<></n; 	t sh { bubl doub i++	hould le[n] ble[n]; +) { . n);	i give	0.	*/																										
1	} delet	ot += ( [i] = i e [] i	out[1]; ;;																															
	delet delet	e [] oi e [] x;	ut; ;																															
Loade Read Loade Read Loade Read Loade Core	ed syn ed syn ing s ed syn ed syn ing s ed syn was	mbols f mbols f ymbols f ymbols f ymbols f ymbols f generat	for /lib from /s for /sci from /l for /lib from /l for /lib ted by	064/ scin inet 11b6 064/ 11b6 064/	/libm het/g 24/li /libc 24/ld /ld_l tests	so.6 pc/co bc.so so.6 l-linu inux-	mpi ile .6.	lers rs/g (r 86-6	5/9 3CC- 10 54.5	cc-4.8 -4.8 debu so.3	4.6 6.1 ugg 2	.1/ /1i ing .(n	11b b64 sy o d	64/ 11 mbo lebu	1ib bgc 1s	gcc c_s fou ng	_s. .so nd) sym	so. .1 	dor ls f	d ne. fou	one nd)		dor	ne.										

#0 0x000000000000000000535 in doConstTest (n=100) at tests.cxx:14 Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6\_3.6.x86\_64 (adb)

(gdb) graph display

Display 1: i (enabled, scope doConstTest, address 0x7fffb0070a2c)

Program terminated with signal 11, Segmentation fault.



#### Workflow

- step steps to following line of code, stepping into functions if necessary
- next goes to next line of code in the current function; doesn't go into subroutines
- print as before
- finish/return finish in this routine, continue from where it was called

<pre>File Edit View Program Commands Status Source Data  i</pre>		•	0							X	DDD	: /h	ome	/s/:	scin	et/	ljdu	rsi,	/bis	ect/	tes	ts.c	xx									
<pre>0 1</pre>		<u>F</u> ile	Edit	View	Prog	ram	Com	mands	Statu	IS .	Sourc	е	Data	l.																		
<pre>tinclude <cstdio> tinclude <cstdio> tinclude <cstdio> tinclude *diffuse.h* /* diffusing a constant should give 0. */ int doconstTest(int n) {     double *in = new double[n];     double * = new double[n];     double * = new double[n];     int[] = 0;     for (int i=0; j(n; i++) {         int[] = 17.;     }     derivative(in, x, out, n);     double tot=0;     for (int i=1; in-1; i++) {         tot += out[1];         x(i] = i;     }     delete [] in:     delete [] out;     delete [] in:     delete []     delete [] in:     delete []     delete []     delete []</cstdio></cstdio></cstdio></pre>	(	(1)																7	<b>D</b> Looku	0 Pi	Ň.	C.	) Iak	GG Uato	) Dh	? Print	 2 play	Plot	- C	2	- <del>()</del> 2010	1
Loaded symbols for /lib64/libm.so.6 Reading symbols from /scinet/gpc/compilers/gcc-4.6.1/lib64/libgcc_s.so.1done. Loaded symbols for /scinet/gpc/compilers/gcc-4.6.1/lib64/libgcc_s.so.1 Reading symbols from /lib64/libc.so.6(no debugging symbols found)done. Loaded symbols for /lib64/libc.so.6 Reading symbols from /lib64/ld-linux-x86-64.so.2(no debugging symbols found)done. Loaded symbols for /lib64/ld-linux-x86-64.so.2		1: 10: #ind #ind /* (int	1 ude 1 docon 1 doub1 1 doub	<pre><cstdi <cmath *diffu ing a stTest e *in t e *out e *out ==0; int i== n[1] = ative( e tot= int i= ot += [i] = e [] i e [] o e [] o</cmath </cstdi </pre>	<pre>o&gt; &gt;&gt; se.h* consta (int n = new new d 0; j<n 17.; in, x, 0.; 1; i<n out[i] i;</n </n </pre>	nt s ) { doub doub ; i+ ; i+	hould le[n] ble[n]; +) { , n); i++)	give j;	0. */																							
		Load Read Load Read Load	led sy ling s led sy ling s led sy ling s led sy	mbols ymbols ymbols ymbols ymbols ymbols	for /1 from for /s from for /1 from for /1	ib64 /sci cine /lib ib64 /lib ib64	/libm net/g t/gpc 64/li /libc 64/ld /ld-1	.so.6 pc/compt bc.so. .so.6 -linux- inux->	mpiler ilers/ .6( (-x86- (86-64	s/g gc no 64.	gcc-4. c-4.6. debug .so.2. 0.2	6.1 1/1 gin	/1ib 1b64 1g sy	64/ //li /mbo	11b bgc 1s	gcc c_s fou ng	_s.: .so nd) sym	so. .1 	1 doni s fe	. dor e.	ne.	d	one.									

0x00000000000400635 in doConstTest (n=100) at tests.cxx:14

Display 1: i (enabled, scope doConstTest, address 0x7fffb0070a2c)

(gdb) graph display i

Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6\_3.6.x86\_64

Compute • calcul

#### Workflow

(gdb) graph display i (gdb)

- If bug has manifested itself, then bug was in first half; rerun, set breakpoint for 1/4 way mark.
- Otherwise, set a new breakpoint for 3/4-way mark.
- Repeat.



Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6\_3.6.x86\_64

∆ Display 1: i (enabled, scope doConstTest, address 0x7fffb0070a2c)

Compute • calcul

#### Pro Tip #I

- Most debuggers let you set conditional breakpoints
- Break in this loop if i > 50.
- gdb:break tests.cxx:
   14 if i > 50
- ddd: option in break pull-down menu

(gdb) graph display (gdb)

△ Display 1: i (enabled, scope doConstTest, address 0x7fffb0070a2c)



#### Pro Tip #2

Loaded symbols for /lib64/ld-linux-x86-64.so.2

(gdb) graph display i

Core was generated by './tests'. Program terminated with signal 11, Segmentation fault.

Display 1: i (enabled, scope doConstTest, address 0x7fffb0070a2c

0x00000000000400635 in doConstTest (n=100) at tests.cxx:14

Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6\_3.6.x86\_64

- Most debuggers let you set watchpoints.
- Break at any line of code if the given variable is changed. eg, watch x
- Variable must be visible from where you are when you set the watchpoint.
- Useful when you know what variable is being mis-set but don't know who's mis-setting it.
- Very useful if you're debugging legacy code with global variables.

0	00				N	DDD: /	home,	/s/sci	inet/l	jdur	si/bi	sect	/tes	ts.cx	×						
Eile	<u>E</u> dit	View	Program	Commands	Status	Source	<u>D</u> ata														
0:	1										Z Looi	) kup l	ñ.	O area	GQ Unit	- ?	Display	Plot	Q H	Ľ,	0
Hir Hir /*	<pre>iclude iclude iclude iclude diffus idoubl doubl int j for (     i     i     deriv doubl for (     t     x } delet delet</pre>	<pre><cstdic <cmath: *diffus ing a ( stTest( e *in = e *x = =0; int i=( n[1] = ative(i int i=1 ot += c [i] = i e [] ou e [] x;</cmath: </cstdic </pre>	<pre>&gt;&gt; ie.h" instant : int n) {     new dou     new doub     rew doub     irt ; in, x, ou     ;; i<n=1; ;;="" i<;;="" i<;;;;;;;;;;;;;;<="" td="" ut[i];=""><td><pre>should give ble[n]; ble[n]; le[n]; ++) { t, n); i++) {</pre></td><td>0. */</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></n=1;></pre>	<pre>should give ble[n]; ble[n]; le[n]; ++) { t, n); i++) {</pre>	0. */																
Loa Rea Loa Rea Loa	ded sy ding s ded sy ding s ded sy	mbols f ymbols mbols f ymbols f mbols f	or /lib6 from /sc or /scin from /lib from /lib6	4/libm.so.6 inet/gpc/comp et/gpc/comp 064/libc.so.6 4/libc.so.6	mpilers/ ilers/gc	gcc-4.6 c-4.6.1 debugg	.1/lib /lib64 ing syn	64/li /libg mbols	bgcc, cc_s. four	_s.s .so. nd).	0.1. 1 do	do ne.	ine.	4.0							



#### **Profiling** Scientific Computing Course, Feb 2013



## Profiling

- Like debuggers for debugging, profilers are evidence-based methods to find performance problems.
- Can't improve what you don't measure.





## Profiling

- Where in your program is time being spent?
- Find the expensive parts
   Don't waste time optimizing parts that don't matter
- Find bottlenecks.

```
case SIM_PROJECTILE:
        vmin = xmin = 0.;
        ymax = xmax = 1.;
        dx = (xmax-xmin)/npts;
        dy = (ymax-ymin)/npts;
        init_domain(&d, npts, npts, KL_NGUARD, xmin, ymin, xmax, ymax);
        projectile_initvalues(&d, psize, pdens, pvel);
        outputvar = DENSVAR;
       break;
/* apply boundary conditions and make thermodynamically consistant */
bcs[0] = xbc; bcs[1] = xbc;
bcs[2] = ybc; bcs[3] = ybc;
apply_all_bcs(&d,bcs);
domain_backward_dp_eos(&d);
domain_ener_internal_to_tot(&d);
/* main loop */
tick(&tt);
if (output) domain_plot(&d);
printf("Step\tdt\ttime\n");
for (time=0.,step=0; step < nsteps; step++, time+=2.*dt) {</pre>
    printf("%d\t%g\t%g\n", step, dt, time);
    if (output && ((step % outevery) == 0) ) {
        sprintf(ppmfilename,"dens_test_%d.ppm", outnum);
        sprintf(binfilename,"dens_test_%d.bin", outnum);
        sprintf(h5filename,"dens_test_%d.h5", outnum);
        sprintf(ncdffilename,"dens_test_%d.nc", outnum);
        domain_output_ppm(&d, outputvar, ppmfilename);
        domain_output_bin(&d, binfilename);
        domain_output_hdf5(&d, h5filename);
        domain_output_netcdf(&d, ncdffilename);
        domain_plot(&d);
        outnum++;
    kl_timestep_xy(&d, bcs, dt);
    apply_all_bcs(&d,bcs);
    kl_timestep_yx(&d, bcs, dt);
    apply_all_bcs(&d,bcs);
tock(&tt);
```

## Profiling

- Tracing vs. Sampling
- Instrumenting vs. instrumentation-free

```
case SIM_PROJECTILE:
       ymin = xmin = 0.;
        ymax = xmax = 1.;
        dx = (xmax-xmin)/npts;
        dy = (ymax-ymin)/npts;
        init_domain(&d, npts, npts, KL_NGUARD, xmin, ymin, xmax, ymax);
       projectile_initvalues(&d, psize, pdens, pvel);
       outputvar = DENSVAR;
       break;
/* apply boundary conditions and make thermodynamically consistant */
bcs[0] = xbc; bcs[1] = xbc;
bcs[2] = ybc; bcs[3] = ybc;
apply_all_bcs(&d,bcs);
domain_backward_dp_eos(&d);
domain_ener_internal_to_tot(&d);
/* main loop */
tick(&tt);
if (output) domain_plot(&d);
printf("Step\tdt\ttime\n");
for (time=0., step=0; step < nsteps; step++, time+=2.*dt) {
    printf("%d\t%g\t%g\n", step, dt, time);
    if (output && ((step % outevery) == 0) ) {
        sprintf(ppmfilename,"dens_test_%d.ppm", outnum);
        sprintf(binfilename,"dens_test_%d.bin", outnum);
        sprintf(h5filename,"dens_test_%d.h5", outnum);
       sprintf(ncdffilename,"dens_test_%d.nc", outnum);
       domain_output_ppm(&d, outputvar, ppmfilename);
       domain_output_bin(&d, binfilename);
        domain_output_hdf5(&d, h5filename);
        domain_output_netcdf(&d, ncdffilename);
       domain_plot(&d);
       outnum++;
    kl_timestep_xy(&d, bcs, dt);
    apply_all_bcs(&d,bcs);
    kl_timestep_yx(&d, bcs, dt);
    apply_all_bcs(&d,bcs);
tock(&tt);
```

compute • calcu

## Timing whole program

- Very simple; can run on any command.
- In serial, real = user + sys
- In parallel, ideally user
   = nprocs x real
- Can run on tests to
   identify performance regressions.

\$ time ./a.out

[ your job output ] Elapsed "walltime" user 0m2.383s sys 0m0.027s System time: Disk, I/O...



#### Watching program run \$ top

top - 2 Tasks:	1:56:45 u 234 total	p	5:56 1 r	, 1 u	iser,	load		averag	ge: 5.5	5, 1.73,	0.8	38 ombie	
Cnu(s):	11.4%us,	36	. 2%5	y, Ø.	0%ni,	52.2	8	id, (	0.0%wa,	0.0%hi,	. (	.2%si, 0.0%st	
Mem: 1	6410900k	tot	al,	15427	68k u	ised,	14	18681	32k free	e,	01	buffers	
Swap:	Øk	tot	al,		Øk u	ised,			0k free	e, 2946	528	cached	
PID U	SER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	Ρ	COMMAND	
22479 l	jdursi	18	0	108m	4816	3212	S	98.5	0.0	1:04.81	6	gameoflife	
22480 l	jdursi	18	0	108m	4856	3260	S	98.5	0.0	1:04.85	13	gameoflife	
22482 1	jdursi	18	0	108m	4868	3276	S	98.5	0.0	1:04.83	2	gameoflife	
22483 l	jdursi	18	0	108m	4868	3276	S	98.5	0.0	1:04.82	8	gameoflife	
22484 1	jdursi	18	0	108m	4832	3232	S	98.5	0.0	1:04.80	9	gameoflife	
22481 l	jdursi	18	0	108m	4856	3256	S	98.2	0.0	1:04.81	3	gameoflife	
22485 l	jdursi	18	0	108m	4808	3208	S	98.2	0.0	1:04.80	4	gameoflife	
22478 l	jdursi	18	0	117m	5724	3268	D	69.6	0.0	0:46.07	15	gameoflife	
8042 r	oot	0	-20	2235m	1.1g	16m	S	2.3	6.8	0:30.59	8	mmfsd	
10735 r	oot	15	0	3702	452	372	S	1 3	0 0	0.16 80	0	cat	

More system then user time not very efficient



## Instrumenting regions of code

- Instrumenting the code
- Simple, but incrediby useful.
- Runs every time your code is run
- Can trivially see if changes make things better or worse

struct timeval calc;

```
tick(&calc);
/* do work */
calctime = tock(&calc);
```

```
printf("Timing summary:\n");
/* other timers.. */
printf("Calc: %8.5f\n", calctime);
```

```
void tick(struct timeval *t) {
    gettimeofday(t, NULL);
```

```
double tock(struct timeval *t) {
   struct timeval now;
   gettimeofday(&now, NULL);
    return (double)(now.tv_sec - t->tv_sec) +
      ((double)(now.tv_usec - t->tv_usec)/1000000.);
```



## Instrumenting regions of code

- Simple example matrix-vector multiply
- Initializes data, does multiply, saves result
- Look to see where it spends its time, speed it up.
- Options for how to access data, output data.

```
/* initialize data */
tick(&init);
gettimeofday(&t, NULL);
seed = (unsigned int)t.tv_sec;
 or (int i=0; i<size; i++) {
       x[i] = (double)rand_r(&seed)/RAND_MAX;
        y[i] = 0.;
 f (transpose) {
    for (int i=0; i<size; i++) {
        for (int j=0; j<size; j++) {
            a[i][j] = (double)(rand_r(&seed))/RAND_MAX;
} else {
    for (int j=0; j<size; j++) {
        for (int i=0; i<size; i++) {
            a[i][j] = (double)(rand_r(&seed))/RAND_MAX;
inittime = tock(&init);
/* do multiplication */
tick(&calc);
(transpose) {
    #pragma omp parallel for default(none) shared(x,y,a,size)
    for (int i=0; i<size; i++) {
        for (int j=0; j<size; j++) {
            y[i] += a[i][j]*x[j];
   3
} else {
    #pragma omp parallel for default(none) shared(x,y,a,size)
    for (int j=0; j<size; j++) {
        for (int i=0; i<size; i++) {
            y[i] += a[i][j]*x[j];
```

CANAD

- Simple example matrix-vector multiply
- Initializes data, does multiply, saves result
- Look to see where it spends its time, speed it up.
- Options for how to access data, output data.

#### \* initialize data \*/ tick(&init); gettimeofday(&t, NULL); seed = (unsigned int)t.tv\_sec; or (int i=0; i<size; i++) { x[i] = (double)rand\_r(&seed)/RAND\_MAX; y[i] = 0.;f (transpose) { for (int i=0; i<size; i++) { for (int j=0; j<size; j++) { a[i][j] = (double)(rand\_r(&seed))/RAND\_MAX; } else { for (int j=0; j<size; j++) { for (int i=0; i<size; i++) { a[i][j] = (double)(rand\_r(&seed))/RAND\_MAX; inittime = tock(&init); /\* do multiplication \*/ tick(&calc); (transpose) { #pragma omp parallel for default(none) shared(x,y,a,size) for (int i=0; i<size; i++) { for (int j=0; j<size; j++) { y[i] += a[i][j]\*x[j]; 3 } else { #pragma omp parallel for default(none) shared(x,y,a,size) for (int j=0; j<size; j++) { for (int i=0; i<size; i++) { y[i] += a[i][j]\*x[j]; CANAD

- Can get an overview of the time spent easily, because we instrumented our code (~12 lines!)
- I/O huge bottleneck.

\$ mvm -	-matsize=2	2500
Timing	summary:	
Init:	0.00952	sec
Calc:	0.06638	sec
I/O :	5.07121	sec



- I/O being done in ASCII
- having to loop over data, convert to string, write to output.
- 6,252,500 write operations!
- Let's try a --binary option:

```
out = fopen("Mat-vec.dat","w");
fprintf(out,"%d\n",size);
for (int i=0; i<size; i++)</pre>
    fprintf(out,"%f ", x[i]);
fprintf(out,"\n",out);
for (int i=0; i<size; i++)</pre>
    fprintf(out,"%f ", y[i]);
fprintf(out,"\n",out);
for (int i=0; i<size; i++) {</pre>
    for (int j=0; j<size; j++) {</pre>
        fprintf(out,"%f ", a[i][j]);
    fprintf(out,"\n",out);
fclose(out);
```



• Let's try a --binary option:

out = fopen("	Mat-vec.dat","w	b");		
fwrite(&size,	<pre>sizeof(int),</pre>	1,	out);	
fwrite(x,	<pre>sizeof(float),</pre>	size,	out);	
fwrite(y,	<pre>sizeof(float),</pre>	size,	out);	
fwrite(&(a[0]	[0]), sizeo	f(float),	size*size,	out);
<pre>fclose(out);</pre>				

• Shorter...



- And much (36x!) faster
- File 4x smaller
- Still slow, but file I/O is always going to be slower than a multiplication.
- On to calculation...

\$ mvm	-matsize=2500
binary	?
Timing s	summary:
Init:	0.00976 sec
Calc:	0.06695 sec
I/O :	0.14218 sec
\$ ./mvm	binary
\$ du -h	Mat-vec.dat
89M	Mat-vec.dat
\$ ./mvm	binary

- \$ du -h Mat-vec.dat
- 20M Mat-vec.dat

## Sampling for Profiling

- How to get finer-grained information about where time is being spent?
- Can't instrument every single line.
- Compilers have tools for sampling execution paths.



## Sampling for Profiling

- As program executes, every so often

   (~100ms) a timer goes
   off, and the current
   location of execution
   is recored
- Shows where time is being spent.



## Sampling for Profiling

- Advantages:
  - Very low overhead
  - No extra instrumentation
- Disadvantages:
  - Don't know why code was there
  - Statistics have to run long enough job



## gprof for sampling



```
$ ./mvm-profile --matsize=2500
[output]
$ ls
Makefile Mat-vec.dat gmon.out
mat-vec-mult.c mvm-profile
```



## gprof examines gmon.out

<pre>\$ gprof mvm-profile gmon.out</pre>									
Flat profile:									
Each sar	mple counts	s as 0.01	seconds.						
% C1	umulative	self		self	total				
time	seconds	seconds	calls	Ts/call	Ts/call	name			
100.24	0.41	0.41	3	0.00		main			
0.00	0.41	0.00	3	0.00	0.00	tick			
0.00	0.41	0.00	3	0.00	0.00	tock			
0.00	0.41	0.00	2	0.00	0.00	alloc1d			
0.00	0.41	0.00	2	0.00	0.00	freeld			
0.00	0.41	0.00	1	0.00	0.00	alloc2d			
0.00	0.41	0.00	1	0.00	0.00	free2d			
0.00	0.41	0.00	1	0.00	0.00	get_options			
[]									

Gives data by function -- usually handy, not so useful in this toy problem

## gprof --line

gpc-f103	n084-\$ gpro	ofline	mvm-p	rofile	gmon.	out   more
Flat pro	ofile:					
Each sam	ple counts	as 0.01 s	second	s.		
% Cu	mulative	self		se	lf	total
time	seconds s	seconds	call	s Ts/	call	Ts/call name
68.46	0.28	0.28		m	ain (m	nat-vec-mult.c:82 @ 401
14.67	0.34	0.06		m	ain (m	nat-vec-mult.c:113 @ 40
7.33	0.37	0.03		m	ain (m	nat-vec-mult.c:63 @ 401
4.89	0.39	0.02		m	ain (m	nat-vec-mult.c:112 @ 40
4.89	0.41	0.02		m	ain (m	nat-vec-mult.c:113 @ 40
0.00	0.41	0.00	3	0.00	0.00	tick (mat-vec-mult.c:159 @ 40
0.00	0.41	0.00	3	0.00	0.00	tock (mat-vec-mult.c:164 @ 40
0.00	0.41	0.00	2	0.00	0.00	alloc1d (mat-vec-mult.c:152 @
0.00	0.41	0.00	2	0.00	0.00	<pre>free1d (mat-vec-mult.c:171 @</pre>
0.00	0.41	0.00	1	0.00	0.00	alloc2d (mat-vec-mult.c:130 @
0.00	0.41	0.00	1	0.00	0.00	free2d (mat-vec-mult.c:144 @
0.00	0.41	0.00	1	0.00	0.00	<pre>get_options (mat-vec-mult.c:1</pre>



# Then can compare to source

80 81

82

83

84

--

98

99 100

101

102

104

106

107 108

109

111

113

115 116

117

- Code is spending most time deep in loops
- #I multiplication
- #2 I/O (old way)

```
for (int j=0; j<size; j++) {</pre>
    for (int i=0; i<size; i++) {</pre>
        y[i] += a[i][j]*x[j];
    }
}
           ...
out = fopen("Mat-vec.dat","w");
fprintf(out,"%d\n",size);
for (int i=0; i<size; i++)</pre>
    fprintf(out,"%f ", x[i]);
fprintf(out,"\n");
for (int i=0; i<size; i++)</pre>
    fprintf(out,"%f ", y[i]);
fprintf(out,"\n");
for (int i=0; i<size; i++) {</pre>
    for (int j=0; j<size; j++) {</pre>
         fprintf(out,"%f ", a[i][j]); 
    }
    fprintf(out,"\n");
7
fclose(out);
```



## gprof pros/cons

- Exists (almost) everywhere
- Easy to script, put in batch jobs
- Low overhead
- As with graphical debuggers, many nice graphical profilers exist as well



#### Mac OS X note

- Sadly, as of 10.5, Mac OS X no longer supports gprof.
- Instruments app in Xcode
  - Open Instruments.
  - Select the "Time Profiler" template.
  - Select your program as the "Target" dropdown menu.
  - Hit the red circle ("record") button.
  - Hit the record button again to stop recording.
  - Use the tools in Instruments to analyze your results.



# Then can compare to source

80 81

82

83

84

--

98

99 100

101

102

104

106

107 108

109

111

113

115 116

117

- Code is spending most time deep in loops
- #I multiplication
- #2 I/O (old way)

```
for (int j=0; j<size; j++) {</pre>
    for (int i=0; i<size; i++) {</pre>
        y[i] += a[i][j]*x[j];
    }
}
           ...
out = fopen("Mat-vec.dat","w");
fprintf(out,"%d\n",size);
for (int i=0; i<size; i++)</pre>
    fprintf(out,"%f ", x[i]);
fprintf(out,"\n");
for (int i=0; i<size; i++)</pre>
    fprintf(out,"%f ", y[i]);
fprintf(out,"\n");
for (int i=0; i<size; i++) {</pre>
    for (int j=0; j<size; j++) {</pre>
         fprintf(out,"%f ", a[i][j]); 
    }
    fprintf(out,"\n");
7
fclose(out);
```



#### Cache Thrashing Cache

- Memory bandwidth is key to getting good performance on modern systems
- Main Mem big, slow
- Cache small, fast
  - Saves recent accesses, a line of data at a time.



#### Cache Thrashing Cache

- When accessing memory in order, only one access to slow main mem for many data points
- Much faster



- When accessing memory out of order, much worse
- Each access is new cache line (cache miss)- slow access to main memory
- Can see ~10x
   slowdown





 In C, cache-friendly order is to make last index most quickly varying



 In C, cache-friendly order is to make last index most quickly varying



- Can see cache problems with valgrind + visualizer:
- valgrind -tool=cachegrind
- KDE tool kcachegrind available for window,s linux, mac os x.

```
Good
/* do multiplication */
tick(&calc);
if (transpose) {
    for (int i=0; i<size; i++) {</pre>
         for (int j=0; j<size; j++) {</pre>
             y[i] += a[i][j]*x[j];
         }
} else {
    for (int j=0; j<size; j++) {</pre>
        for (int i=0; i<size; i++) {</pre>
             y[i] += a[i][j]*x[j];
                                       Bad
    }
calctime = tock(&calc);
```

- Once cache thrashing is fixed, and assuming I/O can't be improved, Init is now the bottleneck!
- So it goes...

\$ ./mvm-omp --matsize=2500 --transpose --binary Timing summary: Init: 0.00947 sec Calc: 0.00811 sec I/O : 0.14881 sec



#### IDEs

- Many choices for IDEs integrated editor, build manager, debugger, profiler.
- Visual Studio, Xcode, Eclipse,..
- Can be **extremely** powerful, useful, especially when learning new language, code base
- Benfits/Costs of integration: have to do everything through IDE.

	C/C++ - Mountains/GenFractal.cpp - Eclipse Platform
<u>File Edit Refactor N</u> avigate Se	earch Project Bun Window Help
🗂 🐨 🗟 🚔 📓 🖆 🚳 🖓	🖻 -
🏷 Project Explorer 🕱 🛛 🗖	GenFractal.cpp 😫
<ul> <li>Image: Second state of the second st</li></ul>	<pre>/************************************</pre>
<ul> <li>main.cpp 1.2 (ASCII -kkv</li> <li>mountains.cpp 1.3 (ASC</li> <li>S &gt;Mountains - [x86/le] 1.:</li> <li>&gt;Makefile 1.7 (ASCII -kko</li> <li>&gt;Makefile.Debug 1.7 (A</li> <li>&gt;Makefile.Release 1.7 (</li> <li>Mountains.pro 1.3 (ASC</li> <li>mountains.ui 1.6 (ASCII</li> <li>README 1.1.1.1 (ASCII -I</li> </ul>	<pre>/************************************</pre>

